MASTER EN SCIENCES INFORMATIQUES

Un interpréteur abstrait Prolog par grammaire attribuée

Gos, P.

Award date:
1994

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES N.D. PAIX NAMUR

INSTITUT D'INFORMATIQUE

Rue Grandgagnage, 21, B-5000 Namur (Belgium)

Un interpréteur abstrait

Prolog par grammaire attribuée

P.Gos

Promoteur : B. Le Charlier

Mémoire présenté pour l'obtention du titre de licencié et maître en informatique

Année académique 93-94

Table des matières.

Ce travail serait incomplet si n'étaient remerciées les personnes qui de près ou de loin ont contribué à son achèvement. Que soit remercié B. Le Charlier qui par sa présence attentive a supervisé la conception et la rédaction de cet écrit. Il serait injuste de ne pas également remercier K. Musumbu pour sa patience et ses conseils avisés. Sa collaboration fut précieuse. Enfin que soient remerciées toutes les personnes qui, par leur présence à mes côtés, ont contribué à me rendre l'existence plus agréable durant la rédaction de ce mémoire.

P. Gos

Abstract.

Abstract interpretation is a static analysis method for programming languages. This subject has been an active field of research for several years and has attracted many researchers. In this report a method of abstract interpretation for Prolog programs is presented. Basics of first order logic, abstract interpretation and attributed grammars are firstly introduced. Follows a brief description of the abstract domain for mode analysis and the related operators for the abstract interpreter. The attributed grammar is then defined. This grammar implements the semantics of the monotonic mapping associated to any Prolog program. It is showed how fixpoint computation can be performed by attribute evaluation for the grammar. In the last chapter, an implementation of the abstract interpreter is presented. Some test programs and an evaluation of the results obtained for these programs are given. Finally some potentially interesting improvement issues are briefly outlined.

Résumé.

L'interprétation abstraite est une méthode d'analyse statique pour langages de programmation. Le sujet a attiré bon nombre de chercheurs et est un domaine de recherche active depuis plusieurs années. Une méthode d'interprétation abstraite pour programmes Prolog est présentée dans ce mémoire. Les bases de la logique des prédicats du premier ordre, de l'interprétation abstraite et des grammaires attribuées sont dans un premier temps introduites. Suit une brève description du domaine abstrait pour l'analyse de modes ainsi que les opérateurs associés pour l'interpréteur abstrait. La grammaire attribuée est ensuite définie. Cette grammaire donne la sémantique de la transformation de fonctions monotone associée à tout programme Prolog. On montre alors comment le calcul de point fixe peut être effectué par évaluation des attributs de la grammaire. Dans le dernier chapitre une implémentation de l'interpréteur abstrait est présentée. Un certain nombre de tests et une évaluation des résultats obtenus pour ceux-ci sont donnés. Enfin, quelques voies d'amélioration potentiellement intéressantes sont suggérées.

INTRODUCTION

Introduction.

A. L'interprétation abstraite : de quoi s'agit-il ?

L'interprétation abstraite est essentiellement une méthode d'analyse statique de programmes. Ce terme recouvre un ensemble de techniques visant à assurer autant que possible la correction et l'optimisation d'un programme donné et cela avant même que son exécution ait lieu. Ces techniques concernent donc non seulement tous les systèmes devant produire du code exécutable à partir d'un langage de haut niveau mais aussi les programmeurs et concepteurs soucieux d'étudier le comportement des programmes qu'ils conçoivent.

L'idée de l'analyse statique de programmes n'est pas véritablement nouvelle. Dès les années 60 existaient déjà des compilateurs capables de vérifier la bonne fin probable du calcul d'une expression ainsi que la spécialisation du code associé selon la technique dite d'analyse de types. Nous utilisons délibérément le terme de technique car, si l'analyse de types est aujourd'hui parachevée et largement diffusée, elle ne constitue néanmoins pas une théorie de l'analyse statique. Un langage tel que Prolog s'il ouvre largement la porte à l'analyse statique ne constitue pas un sujet potentiel pour l'analyse de types ce dernier étant non typé[1].

L'interprétation abstraite résulte d'une tentative (et on l'espère un succès) d'unification et de systématisation des méthodes d'analyse statique de programmes. Elle s'adresse donc à une grande variété de langages de programmation : impératifs, fonctionnels, logiques, orientés objets, ...

Prolog, nous l'avons dit, ne se prête pas à l'analyse de types. Nuançons quelque peu notre propos. La manière classique de faire de l'analyse de types est généralement appelée "type checking". Cette méthode n'est applicable qu'aux langages typés (et donc pas à Prolog). Celle-ci consiste à vérifier si le type d'une expression écrite dans un certain langage est conforme au type requis par le contexte dans lequel elle se trouve. Le type d'une expression étant dérivé à partir de la déclaration des opérateurs et des variables qui la composent. Par exemple, en langage Pascal, on vérifie que les opérandes de l'opérateur "mod" sont bien de type entier.

Une seconde méthode d'analyse de types, appelée "type inference", est applicable à Prolog. Elle consiste à déterminer le ou les types de valeurs possibles pour une variable, un opérateur ou une fonction; l'avantage étant de pouvoir par la suite particulariser le code impliquant ces variables, opérateurs et fonctions. Signalons au passage qu'il est possible de faire de l'inférence de types par interprétation abstraite.

---

[1] Sauf si la notion de type y est introduite artificiellement.

Force est de constater que le code produit par les compilateurs Prolog actuels pèche par son manque d'efficacité : les temps d'exécutions sont supérieurs à ceux des langages procéduraux et les espaces mémoire requis sont très souvent inacceptables. Il s'agit là des conséquences d'une des caractéristiques principales de Prolog : la multidirectionnalité, principe selon lequel les paramètres d'un programme Prolog peuvent être utilisés aussi bien comme données que comme résultats. Une autre caractéristique de Prolog (et de tout langage déclaratif) est que tout programme spécifie les relations qui existent entre les données et les résultats mais ne dit absolument rien sur la manière de calculer ces derniers. Si l'on ajoute à cela le fait que Prolog est un langage non typé et qu'il est dès lors impossible de prévoir à priori la nature des informations à manipuler, on comprend aisément qu'une telle souplesse ne peut être qu'au prix d'un code très général et donc inefficace et volumineux.

La multidirectionnalité offre la possibilité d'un paramétrage libre d'un programme. Dans la pratique on constate que l'utilisateur d'un programme se contente souvent de calculer tels résultats avec telles données. L'interprétation abstraite, tout en conservant la multidirectionnalité, se propose d'analyser le comportement d'un programme donné dans ses différents cas de figure variables/constantes de manière à cerner les cas qui posent problème et à dégager d'éventuelles spécialisations et particularisations du programme considéré.

A la base du principe d'interprétation abstraite se trouvent 3 idées principales. La première consiste à étudier le comportement du programme non pas avec des données réelles mais avec
des représentations abstraites de celles-ci. Par exemple on représentera les nombres entiers non plus par leur valeur mais par leur signe (+, - ou 0). L'analyse consistera alors à étudier le comportement du programme et le signe des résultats en fonction du signe des données.
Cette idée nous conduira à la notion de domaine abstrait. La raison d'une telle méthode est de dégager des informations utiles quant à la nature des résultats de l'exécution d'un programme dans un temps fini (éviter les problèmes de bouclage) et pour bien faire inférieur à celui du calcul sur les valeurs réelles.

Il ne suffit pas de redéfinir les éléments manipulés par un programme il faut également établir un certains nombre d'opérateurs pour les valeurs abstraites. Ainsi dans le cas de la multiplication des nombres entiers on redéfinirait celle-ci par la règle algébrique des signes : +*+=+, +*−=−, +*0=0, −*−=+, ... Il ne s'agit pas d'écrire un équivalent abstrait du programme réel mais bien d'interpréter les opérations qu'il effectue en fonction du domaine abstrait que l'on s'est choisi.

La notion de calcul du plus petit point fixe d'une transformation continue constitue le dernier concept fondateur de l'interprétation abstraite. Toute procédure ou programme peut être représenté par une fonction. Dans le cas présent on considèrera des fonctions de A vers A, A étant le domaine abstrait que l'on s'est choisi. La méthode de calcul de point fixe consiste à construire une suite de fonctions f_i ($\tau(f_i)=f_{i+1}$) telle que chaque fonction de la suite constitue une approximation de la fonction f_p associée au programme P. On montre que moyennant certaines conditions (nous les exposerons par

la suite) concernant le domaine abstrait A et la transformation τ, la suite de fonctions peut être munie d'une relation d'ordre $\leq$ telle que :

$$f_0 \leq f_1 \leq f_2 \leq ..... \leq f_i \leq ....\leq f_p \leq ....$$

avec $f_i \leq f_{i+1}$ signifiant que f_{i+1} donne au moins autant de résultats que f_i . Le plus petit point fixe de τ est défini par $\tau(f)=f$. On peut montrer que la méthode possède trois caractéristiques intéressantes :

1) La fonction f_p fournie par le calcul du plus petit point fixe de τ (et pour laquelle $\tau(f_p)=f_p$) constitue une approximation correcte de la fonction f associée au programme P.

2) La transformation τ possède nécessairement un plus petit point fixe qui constitue la meilleure approximation possible.

3) Le plus petit point fixe de la transformation est la limite de la suite d'approximations que constituent les fonctions f_i.

B. Organisation et structuration du mémoire.

Le texte qui va suivre est articulé en 5 chapitres. Dans un premier temps, nous aborderons la logique des prédicats du premier ordre et la programmation logique en Prolog. Ce sera l'objet du chapitre 1. Après une brève description de la syntaxe de la FOL (First Order Logic), les notions relatives à sa sémantique seront présentées : interprétation, valeur de vérité, modèle, conséquence logique, interprétation de Herbrand, ... La dernière section ainsi que la première du deuxième chapitre seront consacrées à une présentation et une discussion des principes de fonctionnement de Prolog. Les principaux défauts de Prolog seront mis en évidence.

Le second chapitre, sera consacré à une présentation des concepts et principes qui sous-tendent l'interprétation abstraite. Une illustration sera donnée au moyen d'un exemple auquel seront appliqués les concepts présentés.

Au troisième chapitre les grammaires non-contextuelles et attribuées seront introduites. Leur mode d'utilisation sera exposé (dérivation d'une chaîne à partir d'une grammaire) ainsi qu'un mode de représentation graphique associé (arbre syntaxique). Pour les grammaires attribuées, les notions d'attributs synthétisés et hérités seront présentées ainsi qu'une définition générale des règles sémantiques associées à toute grammaire attribuée. Parallèlement nous évoquerons les points critiques des grammaires : unicité d'une dérivation, ambiguïté d'une grammaire non-contextuelle et circularité des règles sémantiques d'une grammaire attribuée.

C'est au quatrième chapitre que sera exposé l'essentiel de la méthode d'interprétation abstraite par grammaire attribuée. Les trois premiers chapitres constituent donc une préparation au propos du quatrième. La structure de cet avant dernier chapitre sera proche de celle du second. On introduira tout d'abord le domaine abstrait utilisé. Les opérateurs abstraits nécessaires seront donnés dans une seconde section. Les trois dernières parties du chapitre seront respectivement consacrées à la grammaire attribuée, l'algorithme de calcul de point fixe et à un exemple destiné à illustrer le fonctionnement de la méthode d'interprétation abstraite présentée.

Enfin, le cinquième et dernier chapitre consistera en une présentation du programme AAIAG[2]. Seront d'abord exposés son mode d'emploi et son architecture globale. Ce programme à été réalisé dans le cadre d'un stage[3] de mémoire effectué au Laboratoire Bordelais de Recherche en Informatique (LaBRI) en collaboration avec K. Musumbu, co-auteur de la méthode présentée au quatrième chapitre. La dernière section du chapitre sera consacrée à quelques exemples d'applications de l'interpréteur abstrait.

[2]An Abstract Intepreter by Attributed Grammar
[3]Effectué d'octobre à décembre 93

CHAPITRE I : LOGIQUE DES PREDICATS DU PREMIER ORDRE ET LANGAGE PROLOG

Chapitre I : logique des prédicats du premier ordre et langage Prolog.

Au cours de ce premier chapitre nous nous proposons de présenter les notions fondamentales nécessaires à la compréhension des chapitres suivants. Nous exposerons d'abord les fondements de la logique des prédicats du premier ordre au travers de sa syntaxe et de sa sémantique. Nous nous limiterons autant que possible aux concepts utiles par la suite tout en mettant l'accent sur les plus importants d'entre eux pour notre propos, à savoir les notions d'interprétation et de modèle ainsi que la sémantique de Herbrand.

Dans un second temps nous nous intéresserons au langage Prolog, à sa syntaxe et à ses principes de fonctionnement. Nous prolongerons cette discussion au début du second chapitre où nous montrerons que Prolog n'est qu'une implémentation imparfaite de la FOL.

A. Logique des prédicats du premier ordre : la syntaxe.

Comme la plupart des langages ou systèmes formels, la FOL possède une syntaxe concrétisée par un alphabet et une grammaire définissant les "phrases" correctes du langage.

En FOL un alphabet est un ensemble de symboles regroupés en 7 catégories :

- les constantes
- les variables
- les symboles de foncteurs
- les connecteurs logiques
- les quantificateurs
- les symboles de ponctuation

En pratique les 4 premières classes de symboles varient d'un alphabet à l'autre alors que les trois dernières sont fixées une fois pour toutes. Comme nous le verrons par la suite une occurrence particulière d'un alphabet sert à la définition d'une théorie du premier ordre.

Les connecteurs logiques quant à eux sont : $\sim$, $\vee$, $\wedge$, $\Leftarrow$ et $\Leftrightarrow$. Les quantificateurs sont : $\forall$ et $\exists$.

Quant aux symboles de ponctuation il s'agit de "(", ")" et ",". De manière informelle la sémantique des connecteurs logiques et des quantificateurs est : "non" ($\sim$), "ou" ($\vee$), "et"

($\wedge$), "équivalent à" ($\Leftrightarrow$), "implique" ($\Leftarrow$), "il existe" ($\exists$) appelé quantificateur existentiel et "pour tout" ($\forall$) également appelé quantificateur universel.

On défini une terme de la manière suivante :

- Une constante est une terme.
- Une variable est un terme.
- Une expression de la forme $f(t_1,t_2, ... ,t_n)$ est un terme si $t_1,t_2, ... ,t_n$ sont des termes et f un symbole de foncteur n-aire.

On appelle atome toute expression de la forme $p(t_1,t_2, ... ,t_n)$ avec p un symbole de prédicat à n arguments et $t_1,t_2, ... ,t_n$ des termes.

On dit d'un symbole de prédicat ou de foncteur qu'il est d'arité n si celui-ci admet n arguments. Un foncteur d'arité nulle est une constante. Un prédicat d'arité nulle est appelé proposition.

Une formule bien formée est définie comme étant :

- soit un atome
- soit une expression de la forme : ~F, F$\vee$G, F$\wedge$G, F$\Leftrightarrow$G ou F$\Leftarrow$G avec F et G des formules bien formées
- soit une expression telle que : $\forall$x F ou $\exists$x F avec F une formule bien formée et x une variable apparaissant dans F.

Ayant défini les concepts nécessaires nous pouvons maintenant donner une définition de ce qu'est un langage du premier ordre. Un langage du premier ordre consiste en l'ensemble des formules bien formées construites à l'aide des symboles d'un alphabet donné.

Venons-en maintenant à la notion de clause.

Définition préalable : un littéral est un atome ou la négation d'un atome (A ou ~A).

Une clause est une formule bien fondée de la forme :

$$\forall x_1,...,x_n \, (L_1 \vee L_2 \vee .... \vee L_m)$$

où $L_1, L_2, ..., L_m$ sont des littéraux et $x_1, ..., x_n$ des variables apparaissant dans $L_1 \vee L_2 \vee .... \vee L_m$.

Signalons au passage qu'il existe une méthode consistant en une série de transformations syntaxiques (mise sous forme normale prenexe, skolemisation et mise sous forme normale conjonctive) qui permettent de ramener toute formule de FOL à une forme clausale. Exposer ici ces méthodes en détail ne nous est pas utile pour la suite.

Il est utile de préciser que les clauses d'un programme ou d'une théorie du premier ordre sont souvent mises sous une autre forme que celle donnée ci-dessus. Soit la clause :

$$\forall x_1,x_2,...,x_n\ (A_1 \vee A_2 \vee .... \vee A_k \vee \sim B_1 \vee \sim B_2 \vee .... \vee \sim B_m) \qquad (1)$$

avec A_1, A_2, ..., A_k et B_1, B_2, ..., B_m des atomes. Si on applique les transformations $P \vee \sim Q$ équivalent à $P \Leftarrow Q$ et $(\sim P \vee \sim Q)$ équivalent à $\sim(P \wedge Q)$ on obtient une forme plus classique pour la clause (1) :

$$\forall x_1,x_2,...,x_n\ (A_1 \vee A_2 \vee .... \vee A_k \Leftarrow B_1 \wedge B_2 \wedge .... \wedge B_m)$$

On désigne par clause de Horn toute clause ayant la forme[1] :

a) $A \Leftarrow B_1 \wedge B_2 \wedge .... \wedge B_n$ $\quad (n \geq 0)$
ou b) $\Leftarrow B_1 \wedge B_2 \wedge .... \wedge B_n$ $\quad (n \geq 1)$

Par la suite nous utiliserons les dénominations suivantes :

- Toute clause de forme (a) avec $n \geq 1$ sera appelée clause.
- Toute clause de forme (a) avec $n=0$ sera appelée un fait.
- Toute clause de forme (b) sera appelée un but.
- Enfin nous appellerons programme tout ensemble non vide de clauses de Horn.

Le dernier concept que nous introduirons pour cette partie consacrée à la syntaxe de la FOL est celui de théorie du premier ordre. Est appelé théorie du premier ordre l'ensemble constitué par un alphabet, un langage du premier ordre, un ensemble de clauses dénommées axiomes plus un ensemble de règles d'inférence (nous expliciterons cette notion par la suite).

Nous en terminerons ici avec la syntaxe de la FOL pour en venir aux aspects, plus importants pour notre propos, liés à sa sémantique.

B. Logique des prédicats du premier ordre : la sémantique.

Nous avons déjà énoncé informellement la sémantique des connecteurs logiques et quantificateurs utilisés en FOL. Nous allons maintenant expliquer ce qu'est la valeur de vérité d'une formule logique, comment on obtient celle-ci et quelle est l'utilité des notions sémantiques.

Nous commencerons par définir la notion d'interprétation. Une interprétation d'un langage du premier ordre est constituée par :

[1]Toutes les variables sont quantifiées universellement.

- Un ensemble non vide D appelé domaine d'interprétation.
- L'affectation d'une valeur de D à chaque symbole de constante du langage.
- Pour chaque foncteur, l'affectation d'une fonction de D^n vers D.
- Et pour chaque symbole de prédicat, une relation de D^n vers {vrai,faux}.

On calcule la valeur de vérité (vrai ou faux) d'une formule de la manière suivante.

Soit I une interprétation d'un langage L muni d'un domaine D. La valeur de vérité d'une formule F écrite dans le langage L est donnée par :

a) Si F est un atome de la forme $p(t_1,t_2,....,t_n)$ alors la valeur de vérité est donnée par la relation $p':D^n \rightarrow \{vrai,faux\}$ associée au prédicat p et les valeurs ($\in D$) des termes $t_1,t_2,....,t_n$ calculées en fonction de I.

b) Si F est de la forme $\sim F_1$, $F_1 \vee F_2$, $F_1 \wedge F_2$, $F_1 \Leftarrow F_2$ ou $F_1 \Leftrightarrow F_2$ alors la valeur de vérité de F est donnée par la table qui suit :

F_1	F_2	$\sim F_1$	$F_1 \vee F_2$	$F_1 \wedge F_2$	$F_1 \Rightarrow F_2$	$F_1 \Leftrightarrow F_2$
V	V	F	V	V	V	V
V	F	F	V	F	F	F
F	V	V	V	F	V	F
F	F	V	F	F	V	V

c) Si F est de la forme $\exists x$ F' alors F a la valeur de vérité vrai si et seulement si il existe $d \in D$ tel que F' a une valeur de vérité égale à vrai lorsqu'on remplace toute occurrence de x dans F' par d.

d) Si F est de la forme $\forall x$ F' alors F a la valeur de vérité vrai si et seulement si la valeur de vérité de F' est vrai lorsqu'on remplace toute occurrence de x dans F' par un quelconque élément de D.

Un concept important pour la suite est celui de modèle. On appelle modèle d'une formule toute interprétation pour laquelle cette formule a une valeur de vérité égale à vrai. De la même manière on dit qu'une interprétation I est un modèle pour S, un ensemble de clauses, si toute formule de S admet la valeur de vérité vrai pour cette interprétation.

Par la suite nous nous intéresserons particulièrement aux conséquences logiques d'un programme. On dira qu'une clause F est une conséquence logique d'un programme P (qui est lui-même un ensemble de clauses) si tout modèle de P est également un modèle de F. Un clause ou ensemble de clauses admettant au moins un (pas de) modèle est dit (in)satisfaisable.

Signalons dès maintenant l'important principe de réfutation qui, comme nous le verrons dans la seconde partie de ce premier chapitre consacrée à Prolog, fait l'objet de l'algorithme de résolution utilisé par ce langage.

Le principe de résolution constitue en fait ce que l'on appelle en mathématiques une démonstration par l'absurde : on nie l'assertion que l'on veut prouver et l'on montre qu'il en découle une contradiction. De manière plus formelle en FOL on dira qu'une formule est une conséquence logique d'un ensemble de formules S si et seulement si $S \cup \{\sim F\}$ est insatisfaisable.

Nous allons maintenant définir les concepts d'interprétation de Herbrand et modèle de Herbrand.

On définit l'univers de Herbrand U_L d'un langage du premier ordre L comme l'ensemble de tous les termes clos (c'est-à-dire formés uniquement de symboles de constantes et de foncteurs) que l'on peut construire avec ce langage L.

La base de Herbrand B_L d'un langage du premier ordre L est l'ensemble de tous les atomes clos de L c'est-à-dire formés uniquement à partir des symboles de prédicats de L et des termes clos de U_L l'univers de Herbrand de L.

Nous pouvons maintenant donner une définition de l'interprétation de Herbrand. Une interprétation de Herbrand d'un langage du premier ordre L est définie comme suit :

- Le domaine d'interprétation est l'univers de Herbrand de L.
- Les constantes sont affectées à elles-mêmes.
- A chaque symbole de foncteur d'arité n est associée une fonction de $(U_L)^n$ vers U_L.
- Enfin à chaque prédicat d'arité n est associée une relation de $(U_L)^n$ vers {V,F}.

On remarque qu'il est possible de caractériser toute interprétation de Herbrand d'un langage du premier ordre comme un sous-ensemble de sa base de Herbrand B_L. Une interprétation de Herbrand peut en effet être définie de manière univoque en donnant l'ensemble des atomes clos qui sont vrai pour celle-ci. Cet ensemble d'atomes constituant évidement un sous-ensemble de B_L.

On qualifiera de modèle de Herbrand d'un ensemble de clauses S toute interprétation de Herbrand de S qui est un modèle pour celui-ci.

Concernant cette notion de modèle de Herbrand, on démontre qu'un ensemble de clauses S est insatisfaisable si et seulement si cet ensemble n'admet pas de modèle de Herbrand.

Nous terminerons là cette section consacrée à la sémantique de la logique des prédicats du premier ordre. Afin d'illustrer les principes et concepts énoncés nous donnons maintenant un exemple.

C. Un exemple.

Considérons l'ensemble de formules FOL qui suit :

Mange_viande(Goliath)
Mange_viande(Max)
Mange_viande(Félix)
Mange_poisson(Félix)
Omnivore(Y) $\Leftarrow$ Mange_viande(Y) $\wedge$ Mange_poisson(Y)
Féroce(Y) $\Leftrightarrow$ race(Y)=Dogue_allemand
Chien(X) $\Leftarrow$ race(X)=Dogue_allemand $\vee$ race(X)=Yorkshire
Chat(X) $\Leftarrow$ race(X)=Persan

L'ensemble des symboles de l'alphabet est :

- Constantes : Goliath, Max, Félix, Dogue_allemand, Yorkshire et Persan.
- Variables : X et Y.
- Foncteurs : race.
- Prédicats : Mange_viande, Mange_poisson, Omnivore, Féroce, Chien, Chat et "=".
- Connecteurs logiques : $\wedge$, $\vee$, $\Leftrightarrow$ et $\Leftarrow$.

Quelques remarques utiles :

- Toutes les formules de l'exemple ci-dessus sont des clauses à l'exception de la sixième d'entre elles. Le symbole d'équivalence "$\Leftrightarrow$" n'étant pas admissible sous forme clausale. Il est néanmoins possible de donner un équivalent sous forme clausale de cette formule. Celle-ci peut en effet être réécrite comme un ensemble de deux clauses :

  Féroce(Y) $\Leftarrow$ race(Y)=Dogue_allemand
  (race(Y)=Dogue_allemand) $\Leftarrow$ Féroce(Y)

- Toutes les formules sont quantifiées universellement : $\forall$ x ... . Il n'y a pas de quantificateurs existentiels.

- Le prédicat "=" est ici utilisé en notation infixée. Une forme plus classique mais moins fréquente consisterait à représenter celui-ci par un prédicat Egal(x,y) (pour x=y).

Illustrons maintenant les concepts sémantiques. Les clauses de notre exemple peuvent admettre une infinité d'interprétations différentes. Il est cependant utile que la sémantique des prédicats et foncteurs utilisés par toute théorie du premier ordre soit en rapport direct avec les noms de ceux-ci. On parle alors d'interprétation première. Dans notre cas il s'agirait de :

Mange_viande(X) ≡ "X mange de la viande"
Mange_poisson(X) ≡ "X mange du poisson"
Omnivore(X) ≡ "X est omnivore"
Chien(X) ≡ "X est un chien"
Chat(X) ≡ "X est un chat"

Quant au foncteur race on le définirait comme :

x	race(x)
Goliath	Dogue_allemand
Félix	Persan
Max	Yorkshire

De manière plus formelle l'interprétation première de notre exemple serait :

- Domaine d'interprétation D = { Goliath, Félix, Max, Dogue_allemand, Yorkshire,Persan }.

- Affectation des constantes :

Constante	d∈ D
Goliath	Goliath
Max	Max
Félix	Félix
Dogue_allemand	Dogue_allemand
Yorkshire	Yorkshire
Persan	Persan

- Foncteur race (D→D) :

X	race(X)
Goliath	Dogue_allemand
Max	Yorkshire
Félix	Persan
Dogue_allemand	Dogue_allemand
Yorkshire	Yorkshire
Persan	Persan

- Prédicat "=" : signification habituelle à savoir l'égalité mathématique.

- Prédicat Mange_viande (D→{V,F}) :

X	Mange_viande(X)
Goliath	V
Max	V
Félix	V
Dogue_allemand	V
Yorkshire	V
Persan	V

- Prédicat Mange_poisson (D→{V,F}) :

X	Mange_poisson(X)
Goliath	F
Max	F
Félix	V
Dogue_allemand	F
Yorkshire	F
Persan	V

- Prédicat Omnivore (D→{V,F}) :

X	Omnivore(X)
Goliath	F
Max	F
Félix	V
Dogue_allemand	F
Yorkshire	F
Persan	V

- Prédicat Féroce (D→{V,F}) :

X	Féroce(X)
Goliath	V
Max	F
Félix	F
Dogue_allemand	V
Yorkshire	F
Persan	F

- Prédicat Chien (D→{V,F}) :

X	Chien(X)
Goliath	V
Max	V
Félix	F
Dogue_allemand	V
Yorkshire	V
Persan	F

- Et enfin le prédicat Chat (D→{V,F}) :

X	(X)
Goliath	F
Max	F
Félix	V
Dogue_allemand	F
Yorkshire	F
Persan	V

On vérifie aisément que cette interprétation constitue un modèle de l'ensemble de clauses donné. En effet, toujours pour cette même interprétation, la valeur de vérité des différentes clauses est la valeur vrai. Vérifions le par exemple pour la clause :

$$\text{Omnivore(Y)} \Leftarrow \text{Mange\_viande(Y)} \wedge \text{Mange\_poisson(Y)}$$

Il suffit pour cela d'appliquer la méthode vue précédemment. Ce qui nous donne :

Y	Mange_viande(Y)	Mange_poisson(Y)	(2) ∧ (3)	Omnivore(Y)	(5) ⇐ (4)
Goliath	V	F	F	F	V
Max	V	F	F	F	V
Félix	V	V	V	V	V
Dogue_allemand	V	F	F	F	V
Yorkshire	V	F	F	F	V
Persan	V	V	V	V	V

L'exemple donné admettant un modèle, on peut dire que l'ensemble de clauses qui le constitue est satisfaisable.

A la fin de la section précédente nous avions énoncé un théorème selon lequel un ensemble de clauses est insatisfaisable si et seulement si cet ensemble n'admet pas de modèle de Herbrand.

Concernant cette notion de modèle de Herbrand nous voudrions illustrer un autre résultat intéressant. Si S est un ensemble de clauses admettant au moins un modèle alors S admet un modèle de Herbrand.

Dans le cas de notre exemple nous avons déjà donné une interprétation qui est un modèle. Donnons maintenant une interprétation de Herbrand qui constitue un modèle de Herbrand. Commençons par construire l'univers de Herbrand et la base de Herbrand pour l'exemple donné.

Univers de Herbrand :

{ Goliath, Max, Félix, Dogue_allemand, Yorkshire, Persan, race(Goliath), race(Max), race(Félix), race(Dogue_allemand), race(Yorkshire), race(Persan), race(race(Goliath)), race(race(Max)), ... }

Base de Herbrand :

{ Mange_viande(Goliath), Mange_viande(Max), ..., Mange_poisson(Goliath), Mange_poisson(Max), Mange_poisson(Félix), ..., Chat(Félix), Mange_poisson(race(Goliath)) , Mange_poisson(race(Max)), ... }

Nous pouvons maintenant construire une interprétation de Herbrand pour l'ensemble de clauses de notre exemple.

- Domaine d'interprétation : U_L l'univers de Herbrand du langage utilisé.

- Affectation des constantes : chaque constante est affectée à elle-même.

- Interprétation du foncteur race ($U_L \rightarrow U_L$) :

x	race(x)
Goliath	race(Goliath)
Max	race(Max)
Félix	race(Félix)
Dogue_allemand	race(Dogue_allemand)
Yorkshire	race(Yorkshire)
Persan	race(Persan)

- Prédicat "=" ($(U_L)^2 \rightarrow \{V,F\}$) :

L'égalité symbolisée par le prédicat égal est définie en donnant la liste des couples (x,y) pour lesquels l'atome "x=y" prend la valeur de vérité vrai :

(race(Goliath),Dogue_allemand), (race(Max),Yorkshire), (race(Félix),Persan).

- Prédicat Mange_viande ($U_L \rightarrow \{V,F\}$) :

X	Mange_viande(X)
Goliath	V
Max	V
Félix	F
Dogue_allemand	V
Yorkshire	V
Persan	F

- Prédicat Mange_poisson ($U_L \rightarrow \{V,F\}$) :

X	Mange_poisson(X)
Goliath	F
Max	F
Félix	V
Dogue_allemand	F
Yorkshire	F
Persan	V

- Prédicat Omnivore ($U_L \rightarrow \{V,F\}$) :

X	Omnivore(X)
Goliath	F
Max	F
Félix	F
Dogue_allemand	F
Yorkshire	F
Persan	F

- Prédicat Féroce ($U_L \rightarrow \{V,F\}$) :

X	Féroce(X)
Goliath	V
Max	F
Félix	F
Dogue_allemand	V
Yorkshire	F
Persan	F

- Prédicat Chien ($U_L \rightarrow \{V,F\}$) :

X	Chien(X)
Goliath	V
Max	V
Félix	F
Dogue_allemand	V
Yorkshire	V
Persan	F

- Prédicat Chat ($U_L \rightarrow \{V,F\}$) :

X	Chat(X)
Goliath	F
Max	F
Félix	V
Dogue_allemand	F
Yorkshire	F
Persan	V

De la même manière que précédemment on peut vérifier qu'il s'agit bien là d'un modèle pour l'ensemble de clauses de notre exemple. Celui-ci vérifie donc bien la propriété énoncée selon laquelle tout ensemble de clauses admettant un modèle admet également un modèle de Herbrand. Remarquons que notre interprétation constitue bien un sous-ensemble de la base de Herbrand. On le vérifie aisément en constatant que l'ensemble des atomes clos ayant la valeur de vérité vrai pour cette interprétation suffit à définir celle-ci. En effet la signification des constantes et le domaine d'interprétation étant fixés une fois pour toutes on peut calculer la valeur de vérité de toute formule du langage L en faisant uniquement référence à la valeur de vérité des différents atomes qui la constituent.

D. Parlons de Prolog.

Développé en 1972 par Alain Colmerauer et son équipe du Groupe d'Intelligence Artificielle de la Faculté des Sciences de Luminy à Marseille, Prolog a été conçu pour être un langage de programmation logique admettant un sous-ensemble de la FOL. Il s'agit bien d'un sous-ensemble car tout programme Prolog est constitué d'un ensemble de clauses de Horn et non de formules de FOL quelconques. Nous donnons ici une définition de la syntaxe de Prolog sous forme d'une grammaire BNF. Il s'agit d'un Prolog générique qui sera à la fois adapté à nos besoins et suffisamment général pour être valable pour n'importe quelle version réelle du langage.

<PRGM>::=<CLAUSE> <PRGM> | <CLAUSE>
<CLAUSE>::=<ATOME>:-<CORPS>. | <ATOME>.
<CORPS>::=<LITTERAL>,<CORPS> | <LITTERAL>
<LITTERAL>::=<ATOME> | not(<ATOME>)
<ATOME>::=<PREDICAT>(<SEQTERMES>)
<SEQTERMES>::=<TERME>,<SEQTERMES> | <TERME>
<TERME>::=<FONCTEUR>(<SEQTERMES>) | <VARIABLE> |
<CONSTANTE> | <LISTE>
<LISTE>::=[<TERME>|<LISTE>] | [<TERME>]

<BUT>::=<CORPS>

Précisons encore qu'un prédicat ou un foncteur est une suite de caractère alphanumériques commençant par une lettre minuscule. Une variable est une suite de caractère alphanumériques commençant par une lettre majuscule. Notez l'apparition de la notion de liste.

Afin de clarifier les choses, donnons l'équivalent Prolog de l'exemple de la section précédente.

```
mange_viande(goliath).
mange_viande(max).
mange_viande(felix).
mange_poisson(felix).
race(goliath,dogue_allemand).
race(max,yorkshire).
race(felix,persan).
omnivore(Y):-mange_viande(Y),mange_poisson(Y).
feroce(Y):-race(Y,dogue_allemand).
race(Y,dogue_allemand):-feroce(Y).
chien(X):-race(X,dogue_allemand).
chien(X):-race(X,yorkshire).
chat(X):-race(X,persan).
```

On remarque plusieurs choses :

- Les constantes sont considérées comme des foncteurs d'arité 0 et donc en respectent la syntaxe.
- Les symboles de conjonction ($\wedge$) et d'implication ($\Leftarrow$) sont représentés par "," et ":-" respectivement.
- Le foncteur race est devenu un prédicat. La relation entre les deux est : race(X,Y) est vrai si et seulement si race(X)=Y.
- L'implémentation directe de l'équivalence n'étant pas possible avec Prolog, la clause "Féroce(Y)$\Leftrightarrow$race(Y)=dogue_allemand" a du être transformée en une double implication.
- Enfin observer comment est traduite la disjonction ($\vee$). Voir les clauses 11 et 12 du programme.

Exécuter un programme Prolog consiste à lui donner un but c'est-à-dire une clause dont le conséquent est vide (voir définition BNF ci-dessus). L'exécution du programme vise à déterminer si le but (la clause du but) est une conséquence logique des axiomes du programme (les clauses de celui-ci). Comment Prolog effectue-t-il cela, c'est ce que nous allons maintenant expliquer.

Deux algorithmes majeurs régissent le fonctionnement de Prolog. Il s'agit d'une part de l'algorithme d'unification et d'autre part de l'algorithme de résolution. L'algorithme d'unification a pour but d'effectuer des instanciations entre les atomes du but courant et les axiomes du programme. L'algorithme de résolution, quant-à-lui, régit l'enchaînement des différentes étapes du processus de résolution, en particulier la sélection des atomes à unifier. Nous allons maintenant décrire l'algorithme d'unification mais auparavant il nous faut définir quelques concepts.

Une substitution est un ensemble fini de la forme $\{v_1/t_1,v_2/t_2,....,v_n/t_n\}$ avec $v_1,v_2,....v_n$ des variables distinctes et $t_1,t_2,....t_n$ des termes ($v_i \neq t_i \ \forall i$).

Soit $\theta=\{v_1/t_1,v_2/t_2,....,v_n/t_n\}$ une substitution et S une expression (soit un terme, soit un atome, soit une conjonction ou une disjonction de littéraux). On appelle instanciation de S par θ (notée $S\theta$) l'expression obtenue en remplaçant dans S toute occurrence de la variable v_i par le terme t_i (i=1,...,n). Par exemple :

$\theta=\{X/f(X),Y/g(X,a)\}$ $\quad\quad$ $S = p(f(X),g(f(z),Y))$

$S\theta = p(f(f(X)),g(f(z),g(X,a)))$

La substitution vide ({}) est notée ε. Elle vérifie la propriété $S\varepsilon=S$ pour toute clause S.

La composition de deux substitutions σ_1 et σ_2, notée $\sigma_2 o \sigma_1$, est une substitution σ appelée composée de σ_1 et σ_2 et est telle que pour toute expression S on ait :

$$S\sigma = (S)(\sigma_2 o \sigma_1) = (S\sigma_1)\sigma_2 = S\sigma_1\sigma_2$$

Voici un exemple :

$\sigma_1=\{X/a,V/g(Y,Z,u)\}$
$\sigma_2=\{X/b,Y/c,Z/f(X),V/h(d),W/f(X)\}$

$\sigma_2 o \sigma_1=\{X/a,V/g(c,f(X),u),Y/c,Z/f(X),W/f(X)\}$

On dit d'une substitution θ qu'elle est un unificateur pour un ensemble d'expressions S si et seulement si $S\theta$ est un singleton c'est-à-dire qu'elle rend les éléments de S identiques.

Un unificateur θ d'un ensemble d'expressions S est appelé unificateur le plus général de S (noté mgu de S) si pour tout unificateur θ' de S il existe une substitution σ telle que $\theta'=\theta\sigma$.

On calcule l'ensemble de discordance d'un ensemble d'expressions S de la manière suivante : comparer parallèlement, symbole à symbole et de gauche à droite les

expressions de S et rechercher la première position de divergence i. L'ensemble des sous-expressions extraites à partir de la position i dans chacune des expressions de S constitue l'ensemble de discordance.

Exemple :

S={p(X,f(g(Y)),b)),p(X,Y,c),p(X,f(Z),b)}

L'ensemble de discordance est {f(g(Y)),Y,f(Z)}

Nous avons maintenant à notre disposition les concepts nécessaires pour présenter l'algorithme d'unification.

ALGORITHME D'UNIFICATION [11] :

$k \leftarrow 0$
$\sigma_0 \leftarrow \varepsilon$
Tant que $S\sigma_k$ n'est pas un singleton faire
    Calculer l'ensemble de discordance D_k de $S\sigma_k$
    Choisir v et t une variable et un terme dans D_k tels que $v \notin t$
    Si c'est impossible alors stop : S n'est pas unifiable
        sinon $\sigma_{k+1} \leftarrow \sigma_k\{v/t\}$
            $k \leftarrow k+1$
    Finsi
Fintantque
Stop : S est unifiable et σ_k est un mgu de S

Quelques commentaires concernant cet algorithme :

- Dans certains cas de figures il existe plusieurs choix possible pour v et t à la ligne 5. Il s'agit donc d'un algorithme non déterministe.
- L'ensemble S étant fini, l'algorithme se termine toujours.
- Les choix v/t effectués n'influencent pas le résultat final si ce n'est par des permutations de variables.

Nous allons maintenant donner un exemple en appliquant l'algorithme d'unification à l'ensemble S={p(Y,g(f(a),Z),b),p(Y,g(X,g(b,a)),b),p(f(b),W,V)}.

Initialisation : $\sigma_0=\varepsilon$.

Itération 1 : $D_0=\{Y,f(b)\}$; $\sigma_1=\{Y/f(b)\}$;
$S\sigma_1=\{p(f(b),g(f(a),Z),b),p(f(b),g(X,g(b,a)),b),p(f(b),W,V)\}$.

Itération 2 : $D_1=\{g(f(a),Z),g(X,g(b,a)),W\}$; $\sigma_2=\{Y/f(b),X/f(a)\}$;

$S\sigma_2$={p(f(b),g(f(a),Z),b),p(f(b),g(f(a),g(b,a)),b),p(f(b),W,V)}.

Itération 3 : D_2={g(f(a),Z),g(f(a),g(b,a)),W}; σ_3={Y/f(b),X/f(a),Z/g(b,a)};
$S\sigma_3$={p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),g(b,a)),b),p(f(b),W,V)}.

Itération 4 : D_3={g(f(a),g(b,a)),W}; σ_4={Y/f(b),X/f(a),Z/g(b,a),W/g(f(a),g(b,a))};
$S\sigma_4$={p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),
g(b,a)),V)}.

Itération 5 : D_4={b,V}; σ_5={Y/f(b),X/f(a),Z/g(b,a),W/g(f(a),g(b,a)),V/b};
$S\sigma_5$={p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),
g(b,a)),b)}.

Itération 6 : Stop.
S est unifiable et σ_5={Y/f(b),X/f(a),Z/g(b,a),W/g(f(a),g(b,a)),V/b} est un mgu.
$S\sigma_5$={p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),g(b,a)),b),p(f(b),g(f(a),
g(b,a)),b)}.

Comme nous allons le voir cet algorithme d'unification est utilisé par l'algorithme de résolution. La résolution consiste en des dérivations successives de faits de plus en plus élémentaires et cela en utilisant les axiomes du programme. L'utilité de l'algorithme d'unification est précisément de faire le lien entre les faits dérivés et les axiomes.

A la base de l'algorithme de résolution l'on trouve la règle de résolution que nous donnons ici.

REGLE DE RESOLUTION (règle d'inférence) :

Soient B_1 et B_2 deux clauses de la forme :

$B_1 : L+ \vee C_1$ $\qquad\qquad$ $B_2 : \sim L- \vee C_2$

avec :
- C_1 et C_2 des clauses.
- L+ et L- des littéraux complémentaires c'est-à-dire ayant le même symbole de prédicat et tels que L+ soit positif (pas de négation) et ~L- soit négatif.
- mgu(L+,L-)=σ.

Si de telles conditions sont vérifiées alors la règle de transition qui suit est applicable.

$$\frac{L+ \vee C_1 \ , \ \sim L- \vee C_2}{(C_1 \vee C_2)\sigma}$$

Nous pouvons maintenant présenter l'algorithme de résolution (extrait de [12]).

ALGORITHME DE RESOLUTION :

Entrées : - Un programme $P=\{A_1,A_2,....,A_n\}$.
- Un but B.

Algorithme:

$S \leftarrow \{A_1,A_2,....,A_n,\sim B\}$
Tant que false$\notin$ S et il existe une paire de clauses résolvables et non résolues
Choisir B_i et B_j dans S et L tel que : $L+\in B_i$, $\sim L-\in B_j$ et $mgu(L+,L-)=\sigma$
$r \leftarrow ((B_i \setminus \{L+\}) \vee (B_j \setminus \{\sim L-\}))\sigma$
$S \leftarrow S \cup \{r\}$
Fintantque
Si false$\in$ S alors B n'est pas une conséquence logique de P
sinon B est une conséquence logique de P

Remarquons que l'algorithme ci-dessus ne fonctionne que si les clauses du programme P sont sous forme clausale (conjonction de littéraux). De ce fait, l'algorithme utilisé par Prolog en est une variante[2] spécialisée. Nous donnons ici un algorithme de résolution pouvant être appliqué à un ensemble de clauses de Horn :

ALGORITHME DE RESOLUTION (clauses) :

Entrées : - Un programme $P=\{A_1,A_2,....,A_n\}$.
- Un but B_0.

Algorithme : Resolution :
$P \leftarrow \{A_1,A_2,....,A_n\}$
$G \leftarrow \{B_0\}$
Resol(G,0)

[2]Rappelons que les clauses d'un programme Prolog ont la forme $A:-L_1,...,L_n$.

```
Resol(G,k):
  stop←faux
  Tant que (G≠∅) et (stop=faux) faire
      Choisir B_i∈G et A_j∈P tels que :
        - le couple (B_i,A_j) n'a pas encore été choisi
        - A_j soit de la forme A_j0:-A_j1,....,A_jn
        - A_j0 et B_i soient unifiables
      Si c'est possible alors
        σ_k←mgu(A_j0,B_i)
        G←G\{B_i}∪{A_j1σ_k,....,A_jnσ_k}
        Si G=∅ alors produire B_0σ_0....σ_k est une conséquence logique
        de P
        Resol(G,k+1)
      Sinon
        stop←vrai
      Finsi
Fintantque
```

Quelques remarques à propos de cet algorithme :

- Il est permis d'utiliser des variables, en particulier pour les buts. Chaque solution consiste donc en une affectation particulière des variables.

- Tout comme le précédent cet algorithme est indéterministe. Aussi l'algorithme utilisé par Prolog est-il différent de celui donné ci-dessus. Lorsqu'une suite de choix pour B_i et A_j donne lieu à un échec, l'algorithme de résolution de Prolog est capable de revenir en arrière et d'effectuer d'autres choix de manière à fournir un ensemble de solutions possibles[3]. Cette capacité de retour en arrière est généralement appelée "backtracking". Présenter cet algorithme ne nous est pas indispensable. Nous montrerons d'ailleurs que les choix effectués par les concepteurs de Prolog pour lever l'indéterminisme de l'algorithme de résolution ont des conséquences assez catastrophiques.

- Cet algorithme utilise l'algorithme d'unification (test si unifiable et calcul de mgu).

Pour en terminer avec ce premier chapitre et clarifier notre propos nous donnons un exemple d'utilisation de l'algorithme de résolution de Prolog. Soit le programme Prolog suivant :

```
nage(X):-poisson(X).
nage(X):-oiseau(X),pattes(X,palmes).
oiseau(donald).
```

[3]En effet, l'ensemble de toutes les solutions possibles n'est pas toujours obtenu

pattes(donald,palmes).

But : nage(Y)

Initialisation : P←{nage(X):-poisson(X).,nage(Y):-
oiseau(Y).,pattes(Y,palmes).,oiseau(donald).,pattes(donald,palmes).}
G←{nage(Y)}
k←0

Etape 1 : G={nage(Y)}, k=0
A_j≡"nage(X):-poisson(X)."
B_i≡"nage(Y)"
σ_0←{X/Y}
G←{poisson(Y)}

Etape 2: G={poisson(Y)}, k=1
Pas de choix possible ⇒ stop.

Etape 3: G={nage(Y)}, k=1
A_j≡"nage(X):-oiseau(X),pattes(X,palmes)."
B_i≡"nage(Y)"
σ_1←{X/Y}
G←{oiseau(Y),pattes(Y,palmes)}

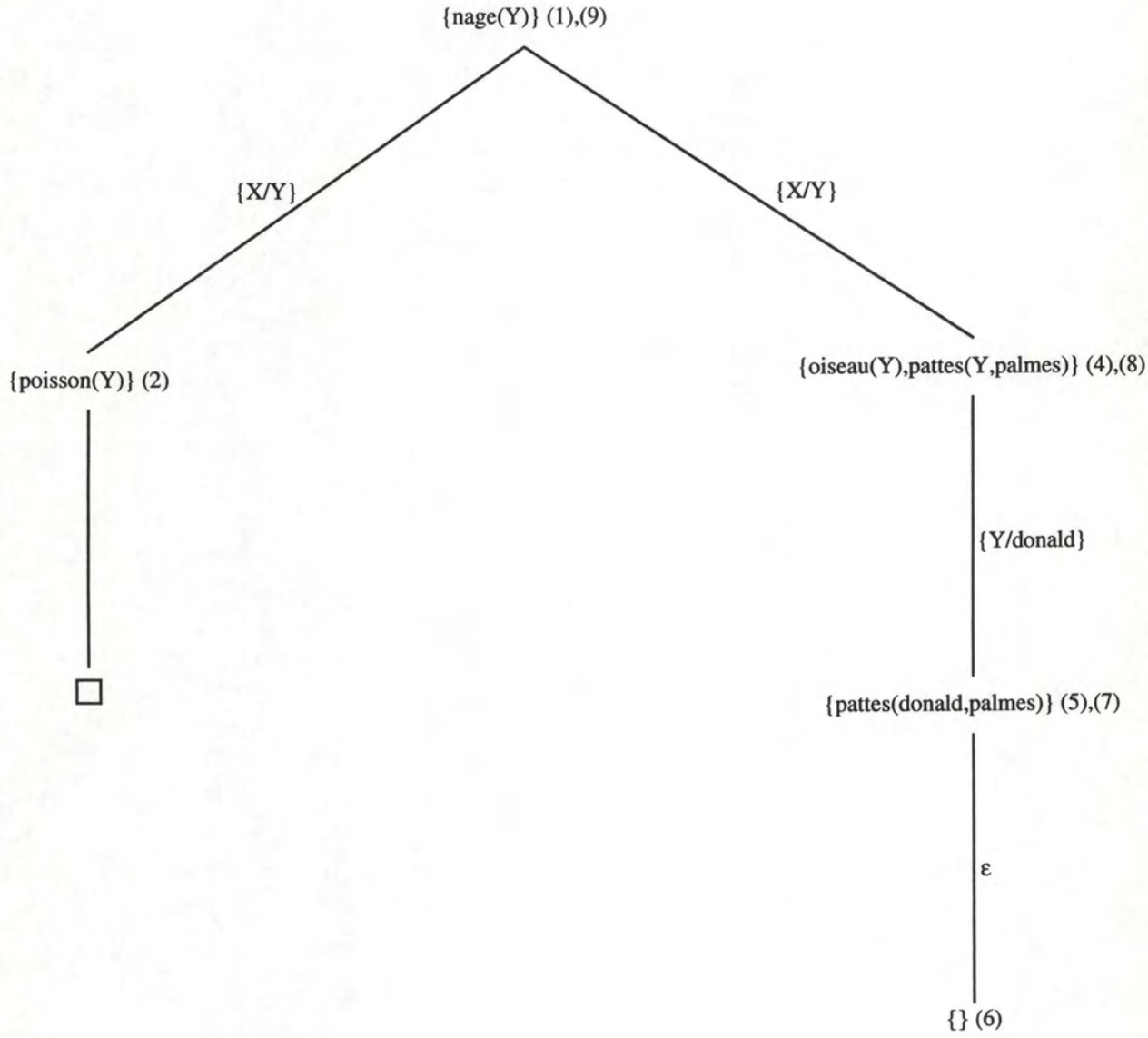
Etape 4: G={oiseau(Y),pattes(Y,palmes)}, k=2
A_j≡"oiseau(donald)."
B_i≡"oiseau(Y)"
σ_2←{Y/donald}
G←{pattes(donald,palmes)}

Etape 5: G={pattes(donald,palmes)}, k=3
A_j≡"pattes(donald,palmes)."
B_i≡"pattes(donald,palmes)"
σ_3←ε
G=∅
⇒ $B_0\sigma_0\sigma_1\sigma_2$ = nage(donald) est une conséquence logique de P.

Etape 6: G=∅, k=4
Stop.

Etape 7: G={pattes(donald,palmes)}, k=3
Pas de choix possible ⇒ stop.

Etape 8: G={oiseau(Y),pattes(Y,palmes)}, k=2
Pas de choix possible ⇒ stop.



Représentation graphique de l'exemple précédent.

Chapitre II : interprétation abstraite.

A. Les maux de Prolog.

Lorsque nous avons présenté l'algorithme d'unification utilisé par Prolog nous avons précisé que les choix variable-terme effectués à chaque itération de l'algorithme n'avaient pas d'influence qualitative sur le résultat fourni. A chaque étape l'algorithme vérifie que la variable choisie v n'apparaît pas dans t, le terme sélectionné. Ce test appelé occur check a une importance non négligeable dans le processus d'unification. Malheureusement, l'algorithme a la fâcheuse caractéristique d'avoir une complexité exponentielle dans certains cas de figure ($o(2^n)$ avec n le nombre total de symboles de l'ensemble à unifier). Des algorithmes à complexité linéaire existent mais les concepteurs de nombreux systèmes Prolog les jugent encore trop coûteux de par la fréquence d'utilisation de l'algorithme d'unification. Ainsi la plupart des compilateurs Prolog n'effectuent pas l'occur check pour des raisons de performance. Un tel choix peut avoir des conséquences graves sur le fonctionnement du système et surtout sur la qualité des résultats produits. Considérons l'exemple suivant emprunté à [11] :

```
test :- p(X,X).
p(X,f(X)) :- p(X,X).
```

Si on donne à Prolog le programme ci-dessus avec le but "test", celui-ci bouclera indéfiniment et ne fournira jamais de réponse (sauf un message d'erreur lorsque la mémoire du système sera pleine). L'absence d'occur check permet en effet l'unification de X avec f(X). La suite des valeurs de l'ensemble de buts pour l'algorithme de résolution est de ce fait infinie et le processus ne s'arrête jamais :

G_0={test}
G_1={p(X,X)}, σ={X/f(X)}
G_2={p(f(X),f(X))} σ={X/f(X)}
.
.
.

Pire encore l'absence d'occur check peut amener le système à fournir des réponses erronées. Modifions quelque peu le programme de l'exemple précédent pour avoir :

```
test :- p(X,X).
p(X,f(X)).
```

L'exécution de ce programme avec le même but "test" fournira la réponse "oui" ce qui tout à fait faux : "test" n'est pas une conséquence logique du programme. De nouveau l'unification de p(X,X) avec p(X,f(X)) pose problème de par l'absence d'occur check.

Quand nous avons présenté l'algorithme de résolution nous avons fait remarquer que celui-ci est indéterministe. La politique choisie dans Prolog pour lever cet indéterminisme est celle du "first fit". Etant donné une occurrence de l'ensemble des buts courants (ensemble G) l'atome sélectionné dans cet ensemble est le premier (le plus à gauche dans le conséquent de la dernière clause utilisée) et il est unifié avec la première clause (selon l'ordre dans lequel les clauses sont écrites dans le programme) avec laquelle il est unifiable.

Remarquons également que l'algorithme de résolution effectue une recherche en profondeur d'abord c'est-à-dire qu'il applique récursivement le processus résolution au but courant. Un tel choix est justifié par le fait que la recherche en profondeur d'abord est facilement implémentable et assez efficace du point vue du temps de calcul et surtout de l'espace mémoire requis.

Malheureusement de tels choix ("first fit" et recherche en profondeur d'abord) ont des conséquences problématiques quant à la complétude du résultat produit par le processus de résolution. Observons le programme qui suit :

```
pere(david,pascal).
pere(pascal,guy).
parent(X,Y):-parent(Y,W),pere(W,X).
```

Si l'on donne comme but "parent(david,guy)" la réponse du système sera "oui" ce qui est correct. Il n'y a donc pas de problèmes dans ce cas. Modifions maintenant le programme comme suit :

```
parent(X,Y):-parent(Y,W),pere(W,X).
pere(david,pascal).
pere(pascal,guy).
```

Celui-ci lancé avec le même but ne s'arrêtera jamais et donc ne fournira pas de réponse alors que la sémantique du programme est absolument la même. Ce problème est dû au fait que l'atome "parent(david,guy)" ne sortira jamais du but courant à cause de la clause "parent(X,Y):-parent(Y,W),pere(W,X)." qui sera toujours sélectionnée avant les deux autres.

Le dernier point faible de Prolog que nous voudrions mettre en évidence est celui du prédicat cut généralement écrit comme "!". L'utilité de celui-ci est de stopper la recherche en profondeur d'abord et de revenir un pas en arrière. Illustrons cela par l'exemple classique de la factorielle d'un nombre.

```
fact(0,1):-!.
fact(X,Y):-Y=X*W,fact(X-1,W).
```

L'effet du prédicat cut est dans ce cas de stopper le processus de résolution (et donc la recherche en profondeur d'abord) lorsque 0! doit être calculé. Si l'on avait supprimé le

prédicat cut et lancé le programme avec le but "fact(4,Z)" celui-ci aurait d'abord répondu "oui Z=24" avant de se lancer dans une boucle infinie. En effet à un moment donné l'unification du but courant "fact(0,Y)" avec "fact(0,1)" aurait produit la réponse donnée ci-dessus mais le processus de résolution aurait ensuite unifié "fact(0,Y)" avec "fact(X,Y):-Y=X*W,fact(X-1,W)" et le but courant serait alors devenu "Y=0*W,fact(-1,W)". Le même phénomène se serait alors poursuivi indéfiniment.

Le prédicat cut permet également au programmeur d'écrire des programmes qui fournissent des résultats corrects mais dont la sémantique est erronée. Pour vous en convaincre considérez l'exemple qui suit [11] :

```
max(X,Y,Y):-X≤Y,!.
max(X,Y,X).
```

D'un point de vue procédural ce programme est correct et calcule effectivement max(X,Y,Z) comme "Z est le maximum de X et Y". D'un point de vue déclaratif et logique le programme ci-dessus est faux, en effet rien ne permet de dire que le maximum de X et de Y est toujours X. C'est pourtant ce qu'affirme la seconde clause du programme. On voit donc que le cut induit un déphasage entre la sémantique opérationnelle et la sémantique déclarative d'un programme. Ajoutons encore à cela que l'existence du prédicat cut ne permet pas d'écrire un programme Prolog correct et efficace tout en ignorant complètement ce qui se passe "derrière" c'est-à-dire la manière dont s'exécute un programme Prolog.

Par cette brève réflexion sur les faiblesses de Prolog nous avons voulu montrer qu'il serait utile de disposer de préprocesseurs capables de mettre en évidence les problèmes liés à un programme donné. C'est précisément là un des but des interpréteurs abstraits.

B. Interprétation abstraite.

L'interprétation abstraite s'appuie sur trois idées fondamentales : le domaine abstrait, les opérateurs abstraits et le calcul de point fixe. De manière intuitive nous dirions que le domaine abstrait et les opérateurs abstraits servent à exécuter un programme donné sur des "valeurs abstraites" au lieu de l'exécuter sur les valeurs habituelles. Le calcul de point fixe quant à lui dirige, à proprement parler, le processus de calcul sur "valeurs abstraites". Comme nous le verrons dans ce qui va suivre, l'interprétation abstraite d'un programme permet d'obtenir des renseignements utiles quant à son exécution et ses résultats. Précisons d'emblée que cela n'est possible que moyennant certaines contraintes relatives au domaine abstrait utilisé. Ainsi on ne peut garantir que le calcul de point fixe soit convergent (temps de calcul fini) que si de telles contraintes ne sont pas respectées.

Dans ce qui va suivre nous présenterons successivement les trois idées fondatrices de l'interprétation abstraite. Dans chaque cas nous définirons les notions théoriques utiles en les illustrant conjointement par un exemple.

B.1. Domaines abstraits.

De manière générale on peut dire que tout programme P écrit dans un langage L quel qu'il soit manipule des données appartenant à un ensemble D que nous appellerons domaine standard. Appliquer l'interprétation abstraite à de tels programmes consiste en un premier temps à construire un ensemble de "valeurs abstraites" A, dérivé de D, et que nous appellerons domaine abstrait. Montrons maintenant comment cela est possible.

Tout programme travaillant avec des données appartenant à un domaine standard D constitue en définitive une fonction de D vers D (où de manière plus générale de D^n vers D^m avec $n,m \geq 0$). Donnons un exemple simple emprunté à [8] : la fonction 91 de Mc Carthy définie comme :

$$f(x)=\text{si } x>100 \text{ alors } x-10 \quad \text{sinon } f(f(x+11)) \qquad (1)$$

Le domaine utilisé est ici Z (D=Z). Nous avons donc une fonction de Z vers Z (ce serait la même chose pour un programme équivalent). Il est bien connu que le calcul de cette fonction pour de grandes valeurs de x est très coûteux en temps de calcul. Pire encore il n'a jamais pu être prouvé que le calcul se termine toujours pour un valeur de x quelconque. Etant donné de telles caractéristiques il pourrait être utile de calculer l'ensemble des valeurs possibles de la fonction pour un ensemble de valeurs de départ donné. Nous aurions donc alors une fonction travaillant non plus sur des valeurs appartenant à Z mais bien sur des éléments de P(Z). Cet ensemble sera appelé domaine concret et noté C. La fonction équivalente à (1) pour le domaine C est alors :

$$f(X)=\{x-10 : x>100 \text{ et } x\in X\} \cup f(f(\{x+11 : x\leq 100 \text{ et } x\in X\}))$$

Cette fonction calcule un ensemble de valeurs à partir d'un autre. Pour un ensemble de départ C_1, la fonction renvoie un ensemble C_2, ces deux ensembles étant tels que $\forall x\in C_1 \; \exists!\, y\in C_2 : y=f(x)$[1]. Malheureusement le calcul d'une telle fonction au moyen d'un programme ad hoc serait bien trop lourd (s'il est déjà trop coûteux pour des valeurs simples il le sera encore bien d'avantage pour des valeurs multiples). De plus, d'un point de vue théorique, les contraintes nécessaires sur le domaine concret C rendraient bien trop complexe la définition des opérateurs abstraits. Que faire dès lors ? Nous allons remplacer le domaine concret C par un domaine abstrait A tel que tout élément de C puisse être approché par un élément de A. Ainsi chaque élément de A constituera donc une approximation pour un certain nombre d'éléments de C.

[1]Pour autant que f(x) soit calculable pour tout $x\in C_1$.

Nous l'avons déjà dit, plusieurs contraintes concernant le domaine abstrait sont nécessaires pour garantir l'efficacité et le caractère praticable du calcul de point fixe. Précisons d'emblée que ces contraintes ne sont pas toujours intégralement requises, selon les cas de figures celles-ci peuvent être affaiblies. Nous allons maintenant préciser ce qu'il faut entendre par contraintes.

Une première caractéristique souhaitable pour le domaine abstrait est que ses éléments véhiculent des informations utiles concernant les éléments du domaine concret qu'ils sont censés représenter. Les éléments de A doivent donc être autant que possible de bonnes approximations des éléments de C. Pour ce faire on exige que soient définies deux fonctions $\alpha : C \rightarrow A$ et $\gamma : A \rightarrow C$ respectivement nommées fonction d'abstraction et fonction de concrétisation pour lesquelles les conditions suivantes doivent être vérifiées :

$$\forall c \in C : \gamma(\alpha(c)) \supseteq c \qquad (2)$$
$$\forall a \in A : \alpha(\gamma(a)) = a \qquad (3)$$

Concrètement la fonction d'abstraction associe à un élément de C sa meilleure approximation dans A. La fonction de concrétisation effectue l'opération inverse c'est-à-dire donne l'ensemble des valeurs de C représentées par un élément déterminé de A. Comme on le voit les conditions (2) et (3) ont pour but principal d'assurer la cohérence des approximations effectuées par le domaine abstrait.

Nous l'avons dit, le calcul d'une fonction sur les valeurs abstraites s'effectue par calcul de point fixe. Pour garantir la terminaison de celui-ci on impose que le domaine abstrait sur lequel on travaille soit ce qu'on appelle un treillis complet. Nous allons expliciter cette notion mais auparavant il nous faut définir une série de concepts indispensables.

Soient un ensemble S et une relation R sur S ($R \subseteq SxS$). On dit de R qu'elle constitue un ordre partiel si et seulement si :

- $\forall s \in S : sRs$
- $\forall s,t \in S : sRt$ et $tRs \Rightarrow s=t$
- $\forall s,t,u \in S : sRt$ et $tRu \Rightarrow sRu$

Par exemple $\leq$ est un ordre partiel pour N.

Soient S un ensemble et $\leq$[2] un ordre partiel sur S. Une borne supérieure de X, un sous-ensemble de S, est un élément $a \in S$ tel que $\forall x \in X : x \leq a$. Semblablement une borne inférieure de X un sous-ensemble de S, est un élément $b \in S$ tel que $\forall y \in X : b \leq y$.

[2] Il s'agit dans ce cas d'une relation d'ordre quelconque et non pas de la relation mathématique habituelle.

On appelle plus petite borne supérieure de X, un sous-ensemble d'un ensemble S muni d'un ordre partiel $\leq$, un élément $s \in S$ tel que s est une borne supérieure pour X et pour toute borne supérieure s' de X on a $s \leq s'$. De la même manière une plus grande borne inférieure de X, un sous-ensemble de S, est un élément t de S tel que t est une borne inférieure pour X et pour toute borne inférieure t' de S on a $t' \leq t$.

On démontre que la plus petite borne supérieure d'un ensemble muni d'un ordre partiel est unique si elle existe. Cette propriété est également vraie pour la plus grande borne inférieure. La plus petite borne supérieure et la plus grande borne inférieure d'un ensemble S sont généralement notées lub(S) et glb(S) respectivement.

Nous pouvons maintenant donner une définition de ce qu'est un treillis complet.

Un ensemble S muni d'un ordre partiel est un treillis complet si et seulement si lub(X) et glb(X) existent pour tout sous-ensemble X de S. On note généralement lub(S) et glb(S) par les symboles $\top$ et $\bot$.

Nous verrons par la suite le rôle que joue cette notion de treillis complet dans le calcul de point fixe. Revenons maintenant à notre exemple de la fonction 91 et montrons comment on peut construire un domaine abstrait pour celle-ci.

Nous avions considéré l'ensemble P(Z) comme domaine concret, il serait dès lors judicieux de donner une approximation de chaque élément $c \in P(Z)$ par un intervalle [min..max] avec min et max les bornes inférieure et supérieure de c. On définit alors le domaine abstrait A comme l'ensemble des intervalles [s..t] avec $s,t \in Z \cup \{-\infty,+\infty\}$ (que nous noterons Z^*). La relation d'ordre sera l'inclusion définie comme [3]:

$[s..t] \subseteq [s'..t']$ si $s \geq s'$ et $t \leq t'$ avec $s,t \in Z^*$

La relation d'inclusion définie comme telle constitue bien une relation d'ordre pour A :

1) $[s..t] \subseteq [s'..t']$ $\forall$ $s,t \in Z^*$
2) $[s..t] \subseteq [s'..t']$ et $[s'..t'] \subseteq [s..t] \Rightarrow [s..t]=[s'..t']$ car $x \geq x'$ et $x' \geq x \Rightarrow x=x'$ $\forall$ $x,x' \in Z^*$
3) $[s..t] \subseteq [s'..t']$ et $[s'..t'] \subseteq [s''..t''] \Rightarrow [s..t] \subseteq [s''..t'']$ de par la propriété de transitivité de la relation $\leq$ pour les entiers.

Ayant construit notre domaine abstrait A nous pouvons montrer qu'il constitue bien un treillis complet en faisant remarquer que :

$$\forall X \subset A : lub(X)=[b_s..b_t] \text{ avec } b_s=\min(\{s \mid \exists t \in Z^* : [s..t] \in X\})$$
$$\text{et } b_t=\max(\{t \mid \exists s \in Z^* : [s..t] \in X\})$$

$$\forall X \subset A : glb(X)=[d_s..d_t] \text{ avec } d_s=\max(\{s \mid \exists t \in Z^* : [s..t] \in X\})$$

[3] $\leq$ et $\geq$ dénotent ici les inégalités mathématiques habituelles.

$$\text{et } d_t=\min(\{t \mid \exists\, s\in Z^* : [s..t]\in X\})$$

[] l'intervalle vide constitue la borne inférieure de A pour l'ordre partiel $\leq$. La borne supérieure étant $[-\infty,+\infty]$.

Les fonctions d'abstraction et de concrétisation sont définies comme :

$\alpha : C\rightarrow A : c\rightarrow\alpha(c)=[\min(c)..\max(c)]$
$\gamma : A\rightarrow C : a\rightarrow\gamma(a)=\{s,s+1,...,t-1,t\}$ avec $a=[s..t]$

On vérifie que les contraintes de cohérence sont bien respectées :

$\forall\, c\in C : \gamma(\alpha(c))\supseteq c$ car $\{\min(c),\min(c)+1,...,\max(c)-1,\max(c)\}\supseteq c$ et
$\forall\, a\in A : \alpha(\gamma(a))=a$ car $\{s,s+1,...,t-1,t\}=[s..t]$ si $a=[s..t]$

Le domaine abstrait que nous venons de construire n'est qu'un exemple. N'importe quel ensemble de valeurs qui constitue une approximation utile et cohérente (vérifiant les contraintes exposées précédemment) du domaine concret utilisé par le programme ou le langage étudié peut servir de base à l'interprétation abstraite de ceux-ci.

Comme vous l'aurez certainement remarqué avoir un domaine abstrait ne suffit pas, encore faut-il disposer d'un équivalent abstrait du programme que l'on veut analyser. C'est précisément là le but des opérateurs abstraits.

B.2. Opérateurs abstraits.

Quand nous parlons d'équivalent abstrait d'un programme, il s'agit d'un programme effectuant les mêmes types d'opérations que l'original mais en manipulant des données appartenant au domaine abstrait utilisé. Pour les besoins de notre exemple nous allons réécrire un équivalent abstrait de la fonction 91. Il est bien évident que lorsqu'il s'agit de faire de l'interprétation abstraite de programmes réels (donc de plus grande taille) écrits dans des langages réels tels que Pascal ou Prolog, effectuer le même travail de réécriture serait long et fastidieux. On trouve là la raison pour laquelle on utilise des opérateurs abstraits; l'interpréteur abstrait pouvant utiliser ceux-ci pour effectuer les calculs sur les données abstraites en interprétant le programme à analyser. Il n'est dès lors pas nécessaire d'écrire un "programme abstrait". En pratique donc, chaque opérateur ou fonction du langage doit posséder un équivalent abstrait.

Concernant ces opérateurs abstraits précisons tout de suite que leur qualité essentielle est leur consistance c'est-à-dire leur cohérence vis-à-vis de leurs équivalents concrets. Ainsi l'approximation d'un résultat réel donnée par un opérateur abstrait devrait toujours être correcte par rapport à celui fourni par l'opérateur concret. Précisons que cette cohérence ne peut, dans certains cas, s'obtenir qu'au prix d'une perte de précision. Une

deuxième qualité importante pour les opérateurs abstraits est que ceux-ci devraient être suffisamment efficaces et convergents pour garantir un temps de calcul fini et acceptable.
Afin de clarifier notre propos revenons à notre exemple. La fonction 91 travaillant sur le domaine C calculait l'ensemble des valeurs résultats en fonction de l'ensemble des valeurs données. De la même manière l'équivalent abstrait de la fonction 91 calcule un intervalle contenant toutes les valeurs de cette fonction pour un intervalle comprenant les valeurs d'entrée de celle-ci. La fonction "abstraite" 91 la voici :

$$f([s..t])=[\max(91,s-10)..(t-10)]\cup f(f([(s+11)..\min(t+11,111)])) \quad (4)$$

Intéressons nous un instant à l'opérateur $\cup$. Sa fonction est de donner une approximation de l'union de deux intervalles par le plus petit intervalle qui les contient. De manière plus précise cet opérateur calcule le lub de deux intervalles :

$$\forall I_1,I_2 \in A \text{ tels que } I_1=[s..t] \text{ et } I_2=[s'..t'] : I_1\cup I_2=\text{lub}(I_1,I_2)=[\max(s,s')..\min(t,t')]$$

On peut facilement se convaincre que l'opérateur est consistant. Si l'ensemble $\{s,s+1,...,t\}\cup\{s',s'+1,...,t'\}$ représente une approximation des valeurs de la fonction 91 pour un intervalle d'entrée donné alors $\text{lub}(I_1,I_2)$ contient au moins les mêmes valeurs que $\{s,s+1,...,t\}\cup\{s',s'+1,...,t'\}$. Nous employons le terme "au moins" car si $\{s,s+1,...,t\}\cap\{s',s'+1,...,t'\}=\varnothing$ alors $\text{lub}(I_1,I_2)$ contient certaines valeurs non contenues dans $\{s,s+1,...,t\}\cup\{s',s'+1,...,t'\}$. Comme nous l'avons dit la consistance se paie souvent au prix d'une perte de précision, c'est le cas ici.

Nous avons construit un domaine abstrait et un équivalent abstrait pour la fonction 91 en vue de calculer des résultats approchés pour celle-ci. Utilisons les et essayons de calculer une approximation des résultats produits par cette fonction. On obtient le calcul ci-dessous :

$$f([-\infty..+\infty])=[91..+\infty]\cup f(f([-\infty..111]))$$
$$f([-\infty..111])=[91..101]\cup f(f([-\infty..111]))$$

On constate malheureusement que le calcul de $f([-\infty..111])$ déclenche récursivement le calcul de $f([-\infty..111])$. La fonction abstraite que nous avons construite est donc inutilisable de manière classique. C'est là que le calcul de point fixe vient à notre secours. Nous allons maintenant montrer comment.

B.3. Calcul de point fixe.

Comme nous allons le voir le calcul de point fixe s'apparente au calcul récursif de fonction. Il nous faut cependant remédier au problème que constitue la terminaison de

tels calculs. Nous allons montrer qu'en procédant par approximations successives on peut obtenir un résultat en un nombre fini d'itérations. Dans ce qui va suivre nous procèderons comme précédemment. Nous présenterons d'abord la notion de plus petit point fixe d'une transformation continue de manière théorique. Nous l'appliquerons ensuite à notre exemple de la fonction 91 en montrant d'une part que la méthode de calcul de point fixe est applicable et d'autre part comment on obtient un résultat à partir de celle-ci. Commençons par donner deux définitions.

Soient A un treillis complet muni d'un ordre partiel $\leq$ et $T : A \rightarrow A$ une transformation[4]. On dit de T qu'elle est monotone si et seulement si $\forall\ x,y \in A : x \leq y \Rightarrow T(x) \leq T(y)$.

On dit d'une transformation $T : A \rightarrow A$ (avec A un treillis complet) qu'elle est continue si et seulement si $T(lub(X)) = lub(T(X))\ \forall\ X \subseteq A$.

Notre intérêt pour les transformations monotones continues vient du fait que le calcul de point fixe consiste essentiellement en l'utilisation d'une transformation de fonctions qui soit monotone continue. Dans la cas de la fonction 91, on raisonne sur l'ensemble des intervalles sur Z^* (le domaine abstrait A). Nous avons montré précédemment qu'il constitue un treillis complet[5]. Nous allons utiliser une transformation de fonctions pour le calcul de point fixe. Cette transformation de fonctions $T : (A \rightarrow A) \rightarrow (A \rightarrow A)$ est la suivante :

$$(Tf)[s..t] = [max(91,s-10)..(t-10)] \cup f(f([(s+11)..min(t+11,111)])) \quad (5)$$

On dit alors que f est un point fixe de la transformation T.

Nous laisserons au lecteur intéressé le soins de montrer que cette transformation vérifie bien les contraintes de continuité et de monotonie ou ce qui revient au même, montrer que la fonction (4) est monotone continue. Il s'agit alors de vérifier les deux conditions ci-dessous :

- $\forall\ I,I' \in A : I \subseteq I' \Rightarrow f(I) \subseteq f(I')$
- Pour toute suite infinie d'intervalles sur Z^* tels que $I_1 \subseteq I_2 \subseteq \ldots \subseteq I_n \subseteq \ldots$[6] on a :

$$f(\cup_{i=1..\infty} I_i) = \cup_{i=1..\infty} f(I_i)$$

L'ensemble des fonctions de $A \rightarrow A$ peut être muni d'une relation d'ordre :

---

[4]Il s'agit en fait d'une fonction comme une autre. Par la suite nous considèrerons des transformations de fonctions : fonctions allant d'un ensemble de fonctions vers lui-même.

[5]Cette contrainte est également nécessaire au calcul de point fixe.

[6]On considère des intervalles emboités précisémment parceque l'application récursive de f génere une suite d'intervalles de tailles croissantes en raison de l'opérateur $\cup$.

$f \leq g$ si et seulement si $\forall I \in A : f(I) \subseteq g(I)$

Si l'on considère f et g comme des fonctions associées à des programmes, on dira alors que le programme calculant la fonction f produit au moins autant de résultats que celui calculant g.

Admettons que l'on veuille calculer f([s..t]) pour s et t donnés, le calcul de point fixe consiste à appliquer de manière récurrente la transformation T à f([s..t]). Concernant cette méthode de calcul de point fixe on démontre les résultats suivants :

- Si les contraintes sur le domaine et les opérateurs abstraits ont été respectées alors tout point fixe de T est une approximation correcte de la fonction f.
- Le plus petit point fixe de la transformation T existe et constitue la meilleure approximation possible de f.
- Le plus petit point fixe de T coïncide avec la limite d'une suite croissante d'approximations :

  $f_0 \leq f_1 \leq \ldots \leq f_n \leq \ldots$

  telle que :

  $f_0(I) = \bot \qquad \forall I \in A$
  $f_{k+1} = T(f_k) \qquad \forall k \geq 0$

Au prix d'un raisonnement long et fastidieux il est possible de donner une expression générale du plus petit point fixe de la transformation (5) et de montrer que la limite de la suite de fonctions engendrée coïncide avec la fonction suivante :

$$f_k([s..t]) = [\max(91,s-10)..\max(91,t-10)] \quad \text{si } t \geq l(k)$$
$$= [] \quad \text{si } t < l(k)$$

avec :

$$l(k) = +\infty \quad \text{si } k=0$$
$$= 102-k \quad \text{si } 1 \leq k \leq 12$$
$$= 222-11*k \quad \text{si } k \geq 13$$

On remarque que l(k) tend vers $-\infty$ lorsque k tend vers $+\infty$. On en déduit que la limite de la suite des f_k (le plus petit point fixe de T) coïncide avec la fonction :

$$f([s..t]) = [\max(91,s-10)..\max(91,t-10)]$$

On peut tirer deux informations intéressantes de ce résultat. D'une part :

$$f([-\infty..+\infty])=[91..+\infty]$$

Ce qui signifie que les valeurs fournies par la fonction 91 sont toutes supérieures à 91. D'autre part on calcule que :

$$f([-\infty..101])=[91..91]$$

Nous l'avions dit précédemment, il n'est pas certain que la fonction 91 soit partout définie. Le résultat ci-dessus nous indique qu'un programme équivalent ayant une valeur d'argument comprise entre $-\infty$ et 101 renvoie 91 ou ne se termine jamais.

Comme vous l'aurez sans doute remarqué la suite d'approximations construite par le calcul de point fixe pour la fonction 91 est infinie. Le résultat ci-dessus (la forme explicite du plus petit point fixe) n'a pu être obtenu que par une analyse spécifique et donc non systématisable. Or notre souhait serait de pouvoir disposer d'une méthode systématique qui nous permettrait de donner des algorithmes de calcul de point fixe. Nous allons maintenant montrer comment on peut y arriver.

B.4. Algorithmes de calcul de point fixe.

Il est utile de faire remarquer que le calcul d'un plus petit point fixe est basé sur une relation de récurrence ($f_{k+1}=T(f_k)$). On se tournera dès lors tout naturellement vers le calcul par récursivité pour les algorithmes désirés. Quand on parle de récursivité ou de définition récursive d'une fonction le premier problème auquel on pense est celui de sa calculabilité. Autrement dit une qualité souhaitable pour les algorithmes de point fixe serait que leur processus de calcul se termine dans tout les cas. La question est comment rendre cela possible. Nous allons voir qu'encore une fois le calcul par approximation va nous venir en aide.

Il existe deux politiques de calcul pour les algorithmes récursifs : le calcul ascendant et le calcul descendant. La première consiste à commencer par calculer les valeurs immédiatement calculables puis celles qui dépendent uniquement de celles-la et ainsi de suite. Par exemple pour calculer 8! on calculerait d'abord 1! puis 2!=2*1!, puis 3!=3*2!, . . . et enfin 8!=8*7!.
Il s'agit de la méthode ascendante que l'on appelle généralement méthode "bottom-up".

Pour une fonction f quelconque une telle manière de faire possède malheureusement le désavantage de calculer un ensemble de couples (a,f(a)) dont certains ne sont pas nécessaires au calcul du résultat que l'on souhaite obtenir. Une méthode plus efficace serait alors de déterminer la suite des valeurs de la fonction qui soit absolument nécessaires pour obtenir le résultat voulu. Malheureusement déterminer dynamiquement une telle suite de valeurs est souvent d'une complexité inacceptable pour être utilisable

dans la pratique (n'oublions pas que notre but est d'obtenir des algorithmes, la méthode doit donc être systématique et implémentable).

Appliquée à l'interprétation abstraite d'une fonction f quelconque la méthode ascendante procéderait du calcul d'une suite d'approximations du plus petit point fixe :

$f_0(a)=\perp \quad \forall\, a\in A$
$f_{k+1}=T(f_k) \quad \forall\, k\geq 0$
$f\equiv f_n \quad$ avec $f_{n+1}=f_n$

La seconde méthode de calcul de fonction récursives, la méthode descendante également appelée "top-down", consiste à effectuer le calcul des valeurs de la fonction dont on a immédiatement besoin pour calculer celle que l'on veut obtenir. On procède ensuite de la même manière pour les nouvelles valeurs nécessaires et ainsi de suite jusqu'à ce que l'on puisse effectuer les calculs de manière directe. Les résultats sont ensuite répercutés sur les étapes précédentes. Le calcul descendant a l'avantage d'être aisément systématisable : à tout moment on déduit les calculs nécessaires de la définition même de la fonction. Cette méthode a cependant le désavantage de calculer plusieurs fois la même chose dans certains cas. Prenons l'exemple de la suite de Fibonnacci :

f(t)=si t∈ {0,1}
alors x
sinon f(x-1)+f(x-2)

On remarque que le calcul de f(b) pour un b quelconque supérieur à 2 déclenchera le calcul multiple de certaines valeurs de f. On peut remédier à un tel problème en utilisant une technique de memo-ization.

La memo-ization consiste à mémoriser dans une table chacune des valeurs de la fonction qui ont déjà été calculées, lorsqu'on doit effectuer le calcul de f(x) pour un certain x on vérifie en consultant la table que celui-ci n'a pas déjà été effectué. Si c'est le cas, on extrait le résultat voulu de cette table. Dans le cas contraire on effectue un appel récursif à la fonction.

C'est cette méthode de calcul descendant avec memo-ization qui sera utilisé dans l'interpréteur abstrait pour programmes Prolog que nous présenterons au chapitre 4. Afin de bien comprendre de quoi il s'agit appliquons la méthode en question à l'interprétation abstraite de la fonction 91. Rappelons que la transformation utilisée est :

$$(Tf)([s..t])=[\max(91,s-10)..(t-10)]\cup f(f([(s+11)..\min(t+11,111)]))$$

Notre démarche nous avait précédemment amené à calculer $f([-\infty..+\infty])$. Nous allons effectuer ce même calcul en considérant au départ que $f(I)=[]\ \forall\ I\in A$. Chaque fois que nous devrons calculer une approximation de f([s..t]) avec s et t donnés, nous utiliserons pour cela les meilleures approximations obtenues jusque là pour les résultats dont nous

avons besoin. On obtiendra alors une première approximation par défaut du plus petit point fixe de T. On effectuera alors le même calcul en raffinant les approximations obtenues à l'étape précédente et ainsi de suite jusqu'à ce qu'on observe plus d'amélioration pour la valeur de f([s..t]) que l'on voulait calculer. En pratique cela donne :

f([−∞..+∞]) =[91..+∞]∪f(f([−∞..111]))
=[91..+∞]∪f([]) car on suppose que f(I)=[] ∀ I∈A
=[91..+∞]∪[]
=[91..+∞]

f([−∞..111]) =[91..101]∪f(f([−∞..111]))
=[91..101]∪f([])
=[91..101]∪[]
=[91..101]

⇒ f([−∞..+∞]) =[91..+∞]∪f([91..101])
=[91..+∞]∪[] car f([91..101]) n'ayant jamais été calculé on suppose que sa valeur est égale à []
=[91..+∞]

Même si l'on n'a pas obtenu d'amélioration pour l'approximation de f([-∞..+∞]), on n'est pas encore au plus petit point fixe car on peut encore espérer améliorer notre résultat approché en raffinant l'approximation de f([91..101]) :

f([91..101]) =[91..91]∪f(f([102..111]))
=[91..91]∪f([])
=[91..91]

⇒ f([−∞..+∞]) =[91..+∞]∪[91..91]
=[91..+∞]

Essayons encore de raffiner notre approximation de f([102..111]) :

f([102..111]) =[92..101]∪f(f([113..111]))
=[92..101]∪f([])
=[92..101]∪[]
=[92..101]

Répercutons ce résultat sur celui de f([91..101]) :

f([91..101]) =[91..91]∪f([92..101)
=[91..91]∪[]

$$=[91..91]$$

$$\Rightarrow \quad f([-\infty..+\infty]) = [91..+\infty] \cup [91..91]$$
$$= [91..+\infty]$$

L'étape suivante serait d'améliorer la valeur approchée de f([92..101]). Après de longs calculs, on obtiendrait finalement que :

$$f([-\infty..+\infty]) = [91..+\infty]$$

Ce qui est en fait la meilleure approximation possible en utilisant notre méthode. On constate également qu'elle est identique à celle donnée par le résultat théorique que nous avions obtenu auparavant. Il ne faut cependant pas perdre de vue que cette méthode de calcul de point fixe fournit une approximation du plus petit point fixe. Dans l'exemple ci-dessus, il se fait que le résultat réel et son approximation coïncident; ce ne sera pas toujours le cas.

B.5. Terminaison de l'algorithme de calcul de point fixe.

Dans l'exemple de la fonction 91 le domaine abstrait est un ensemble infini. Etant donné une telle caractéristique, il est possible que l'algorithme de calcul de point fixe boucle. Cela se produira lorsque l'algorithme effectue une infinité d'appels récursifs différents c'est-à-dire, lorsqu'une infinité de valeurs de la fonction abstraite est requise. Même si dans le cas de la fonction 91 cela ne se produira jamais[7], on ne peut affirmer qu'il en sera toujours de même en général. On pourrait se limiter à des domaines abstraits de taille finie mais dans certains cas cela constituerait une contrainte gênante voire inacceptable. Une voie de solution possible serait d'effectuer le calcul de point fixe non pas au moyen d'une suite infinie d'approximations mais bien en remplaçant cette infinité de valeurs par un nombre fini de valeurs approchées. Par exemple dans le cas du calcul de f([−∞..+∞]) dans notre exemple on pourrait remplacer l'appel récursif à f([−∞..111]) par f([−∞..+∞]). Cela resterait cohérent puisque $[-\infty..111] \subseteq [-\infty..+\infty]$ mais impliquerait une perte de précision. Une telle manière de faire nous permettrait en outre d'accélérer le calcul du plus petit point fixe en réduisant le nombre d'itérations de l'algorithme.

Nous ne nous intéresserons pas d'avantage à cette méthode car le domaine abstrait utilisé par l'interpréteur abstrait Prolog que nous présenterons utilise un domaine fini[8]. Le cas de la non convergence de l'algorithme de calcul de point fixe ne nous inquiétera donc pas d'avantage.

---

[7]On remarque que l'intervalle [(s+11)..min(t+11,111)] qui fait l'objet de l'appel récursif dans le calcul de f([s..t]) est de taille strictement décroissante (s+11 étant strictement croissant et min(t+11,111) borné supérieurement par la valeur 111). Il arrivera donc toujours un moment où cet intervalle deviendra égal à [] ce qui mettra fin à la récursion.

[8]Bien que l'on puisse modifier les opérateurs abstraits de la grammaire attribuée pour qu'ils travaillent sur un domaine abstrait non fini.

Notre propos sur l'interprétation abstraite se termine ici. Nous nous y intéresserons de nouveau au quatrième chapitre consacré à l'interprétation abstraite de programmes Prolog. Il est maintenant temps de consacrer quelques lignes à la seconde pierre d'angle de cet interpréteur : les grammaires attribuées.

CHAPITRE III : GRAMMAIRES NON CONTEXTUELLES ET ATTRIBUEES

Chapitre III : grammaires non contextuelles et attribuées.

Deux éléments différents sont à la base de l'interpréteur abstrait qui sera présenté au quatrième chapitre : l'interprétation abstraite, nous venons d'en parler, et les grammaires attribuées qui vont faire l'objet du présent chapitre. Dans un premier temps, nous nous intéresserons seulement à la nature syntaxique des grammaires. Nous définirons ce qu'est une grammaire non contextuelle. Nous montrerons comment un langage peut être défini par une grammaire et comment on dérive les différentes chaînes de ce langage à partir de cette grammaire. Nous donnerons également deux types de représentation graphique pour une dérivation. Enfin nous évoquerons le problème de l'ambiguïté d'une grammaire. Nous appuierons notre propos par plusieurs exemples.

Dans un second temps nous introduirons les grammaires attribuées comme moyen de définition de la sémantique d'un langage engendré par une grammaire non contextuelle. Nous présenterons les notions importantes d'attributs synthétisés et hérités pour lesquelles nous donnerons à nouveau un exemple. La vérification de la cohérence d'une grammaire attribuée et le mode de calcul de ses attributs seront abordés. Nous définirons pour cela ce qu'est le graphe de dépendances d'une grammaire attribuée et comment il peut être utilisé pour déterminer l'ordonnancement du calcul des attributs.

A. Grammaires non contextuelles.

A l'origine les grammaires non contextuelles furent principalement utilisées pour spécifier la syntaxe des langages de programmation et servir de base à la phase d'analyse syntaxique des compilateurs. Définir la syntaxe d'un langage est encore aujourd'hui l'utilisation première des grammaires non contextuelles. Comme nous le verrons plus tard les grammaires attribuées permettent d'en élargir encore le champ d'application : tout processus de traitement lié à l'analyse d'un langage peut être en quelque sorte "dirigé par la syntaxe" au moyen d'une grammaire attribuée[1] adaptée. Avant d'en arriver là, il nous faut dire ce qu'est une grammaire non contextuelle.

A.1. Définition.

De manière formelle une grammaire non contextuelle est définie par un quadruplet G=(S,N,A,P) avec S, N et P des ensembles finis et A un symbole appartenant à N :

- S est l'ensemble (fini) des symboles de la grammaire et est partitionné en deux classes : les symboles terminaux et les symboles non-terminaux. Les terminaux

[1]Pour être tout à fait préçis deux types de constructions différentes quant à leur niveau d'abstraction peuvent être utilisées : les définitions dirigées par la syntaxe et les schémas de traduction.

sont les symboles de base de la grammaire, c'est à partir d'eux que les chaînes du langage sont formées. Les symboles non-terminaux quant à eux sont en quelque sorte des variables syntaxiques. Ils définissent un ensemble de chaînes qui est un sous-ensemble du langage défini par la grammaire.

• N est l'ensemble des symboles non-terminaux. L'ensemble des symboles terminaux est donc donné par S\N.

• A∈N est appelé symbole distingué ou axiome. Les chaînes de symboles qu'il définit constituent l'ensemble des expressions ou "phrases" valides du langage.

• P est un ensemble fini d'expressions de forme $Y_0 \rightarrow Y_1 Y_2 ... Y_n$ ($n \geq 0$) appelées productions. Y_0 est un symbole non-terminal ($Y_0 \in N$) et $Y_1, Y_2, ..., Y_n$ des symboles appartenant à S.

Donnons maintenant un exemple. Voici comment on peut définir la syntaxe de la numérotation romaine au moyen d'une grammaire non contextuelle :

nr*→*cm cc cd cu
cm→MMM | MM | M | ε
cc→CM | DCCC | DCC | DC | D | CD | CCC | CC | C | ε
cd→XC | LXXX | LXX | LX | L | XL | XXX | XX | X | ε
cu→IX | VIII | VII | VI | V | IV | III | II | I

Avant de montrer comment on dérive un chaîne valide d'un langage à partir d'une grammaire le définissant faisons quelques commentaires concernant la grammaire ci-dessus.

- L'ensemble des symboles S du langage des nombres romains est {***nr***,***cm***,***cc***,***cd***,***cu***,M,D,C,L,X,V,I}. L'ensemble des symboles non-terminaux N est {***nr***,***cm***,***cc***,***cd***,***cu***}. L'ensemble P comporte 5 productions définissant les symboles non-terminaux de N. L'axiome est ***nr***.

- La logique de la grammaire est simple. Chaque chiffre arabe (1,2,3,...,9) est réécrit selon la numérotation romaine et cela pour chaque puissance de 10 allant 0 à 3 (productions 2 à 5). Le cas des milliers est particulier puisque le même chiffre romain ne peut se trouver dans la représentation d'un nombre plus de trois fois de suite consécutivement et qu'il n'y a pas de chiffre de valeur supérieure à 1000. Rappelons que les valeurs des chiffres romains sont : I=1, V=5, X=10, L=50, C=100, D=500 et M=1000.

- Notons la présence de deux symboles particuliers : | et ε qui n'apparaissent pas dans la définition que nous avons donnée. Le premier "|" est en quelque sorte un opérateur "ou" pour les productions. L'ensemble des productions définissant un même symbole non-terminal peuvent être ainsi regroupées en une seule. Par

exemple la première production de notre exemple s'écrirait classiquement comme :

nr→cm
nr→cc
nr→cd
nr→cu

L'utilité essentielle et unique de cet opérateur est de permettre une écriture plus compacte d'une grammaire. Sans lui notre grammaire des nombres romains aurait comporté 37 productions au lieu de 5.

Le second symbole : "ε" a pour but de représenter la chaîne vide c'est-à-dire la chaîne ne comprenant aucun symbole. Son utilité dans l'exemple donné est évidente : un nombre pouvant se limiter aux unités, les chiffres représentant les milliers, les centaines et les dizaines n'apparaîtront donc pas toujours dans la représentation romaine d'un nombre.

- Tout au long du présent chapitre nous utiliserons comme convention de représenter les symboles non-terminaux en caractères gras italiques. Le symbole distingué sera toujours défini en premier.

A plusieurs reprises nous avons utilisé les expressions "dérivation d'une chaîne par une grammaire" ou "définition d'un ensemble de chaînes par un symbole non-terminal". Précisons maintenant ce qu'il faut entendre par là.

A.2. Dérivation d'une chaîne à partir d'une grammaire.

Le point de vue dérivationnel associé à une grammaire consiste à considérer une production comme un règle de réécriture pour laquelle le non-terminal défini est remplacé par la suite de symboles le définissant dans la production utilisée. Par exemple si on a la production ***add→opérande+expr*** celle-ci nous indique que le symbole ***add*** peut être remplacé dans toute chaîne de symboles par ***opérande+expr***. Dans une telle optique, le symbole → signifie "peut avoir la forme de"[2].

Considérons la grammaire suivante :

expr→expr+expr | ***expr**** ***expr*** | (***expr***) | −***expr*** | ID

On peut appliquer répétitivement les productions de cette grammaire pour obtenir la chaîne
"-(ID)" :

[2]N'oublions pas que plusieurs productions différentes peuvent définir un même symbole.

expr $\Rightarrow$ ***−expr*** $\Rightarrow$ ***−(expr)*** $\Rightarrow$ −(ID)

Pour ce faire, on a appliqué successivement les productions ***expr*→*−expr***, ***expr*→*(expr)*** et ***expr*→**ID. On appelle dérivation de ***expr*** en −(ID) une telle séquence de remplacements d'une chaîne par une autre. De manière générale, on peut décrire cela en disant que $\alpha A\beta \Rightarrow \alpha\gamma\beta$ si $A \rightarrow \gamma$ est une production de la grammaire considérée et si α et β sont des chaînes de symboles (terminaux ou non) de cette même grammaire. Si $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ on dira que α_0 "se dérive" en α_n. Une notation fréquemment utilisée consiste à écrire $\alpha_0 \overset{*}{\Rightarrow} \alpha_n$ ce qui signifie que α_0 se dérive en α_n en zéro ou plusieurs étapes. La dérivation vérifie les propriétés suivantes :

1) $\alpha \overset{*}{\Rightarrow} \alpha$ pour toute chaîne α quelconque

2) Si $\alpha \overset{*}{\Rightarrow} \beta$ et $\beta \Rightarrow \gamma$ alors $\alpha \overset{*}{\Rightarrow} \gamma$

De la même manière, on utilise la notation $\overset{+}{\Rightarrow}$ pour signifier "se dérive en une ou plusieurs étapes". Typiquement, on dira pour une grammaire G de symbole distingué ***s*** que $s \overset{+}{\Rightarrow} \omega$ pour tout $\omega \in L(G)$, L(G) étant le langage défini par la grammaire G. On trouve là une définition du langage engendré par une grammaire donnée.

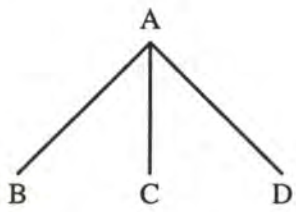
Remarquons qu'à chaque étape d'une dérivation il faut choisir le non-terminal que l'on va remplacer et la production associée que l'on va utiliser. Le lecteur attentif aura constaté une certaine ressemblance avec le processus de résolution utilisé par Prolog : chaque clause d'un programme constitue en quelque sorte une production et chaque atome du but courant un symbole non-terminal. Le but initial étant l'équivalent de l'axiome.

Voici un exemple de dérivation pour notre grammaire de la numérotation romaine. Nous montrons comment la représentation "romaine" de 1994 se dérive de l'axiome ***nr*** en donnant à chaque fois la production utilisée. Le symbole remplacé sera toujours le plus à gauche de la chaîne courante.

nr	***nr*→*cm cc cd cu***
cm cc cd cu	***cm*→**M
M ***cc cd cu***	***cm*→**CM
MCM ***cd cu***	***cd*→**XC
MCMXC ***cu***	***cu*→**IV
MCMXCIV	

A.3. Arbres syntaxiques.

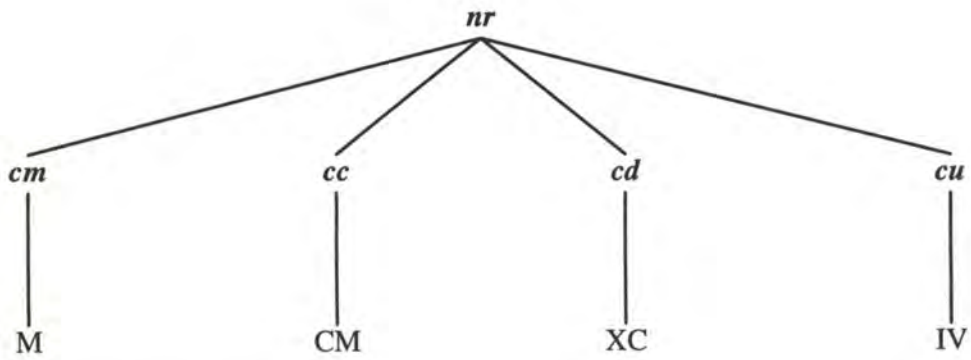
Il est souvent utile et agréable de disposer d'une représentation graphique d'une dérivation. L'arbre syntaxique d'une dérivation permet de visualiser comment l'axiome ou un symbole non-terminal particulier se dérive en une chaîne donnée. Le principe consiste à représenter l'application des productions d'une dérivation en construisant un arbre équivalent. Par exemple, si une production $A \rightarrow B\ C\ D$ est utilisée on représentera cela par :



De manière plus précise un arbre syntaxique pour une dérivation donnée possède les propriétés suivantes :

- La racine est étiquetée par le symbole distingué.
- Chaque feuille est étiquetée par un symbole terminal ou par ε.
- Chaque noeud intérieur (qui n'est pas une feuille) est étiqueté par un symbole non-terminal.
- Si A est l'étiquette d'un noeud intérieur et si les successeurs de ce noeud sont étiquetés par $B_1, B_2, ..., B_n$ alors $A \rightarrow B_1\ B_2\ ...B_n$ est une production de la grammaire. Dans le cas où la production utilisée est $A \rightarrow \varepsilon$ alors le noeud correspondant au symbole A a un seul successeur étiqueté par ε.

A titre d'exemple nous donnons l'arbre syntaxique pour la dérivation de 1994 selon la grammaire "romaine"[3] :



[3]Pour être tout à fait exact ce n'est pas "1994" qui est dérivé mais la chaine MCMXCIV. La valeur 1994 étant en quelque sorte la sémantique décimale de MCMXCIV.

On remarque que les feuilles de l'arbre lues de gauche à droite constituent la chaîne engendrée par la dérivation représentée. On appelle la chaîne ainsi formée le mot des feuilles de l'arbre.

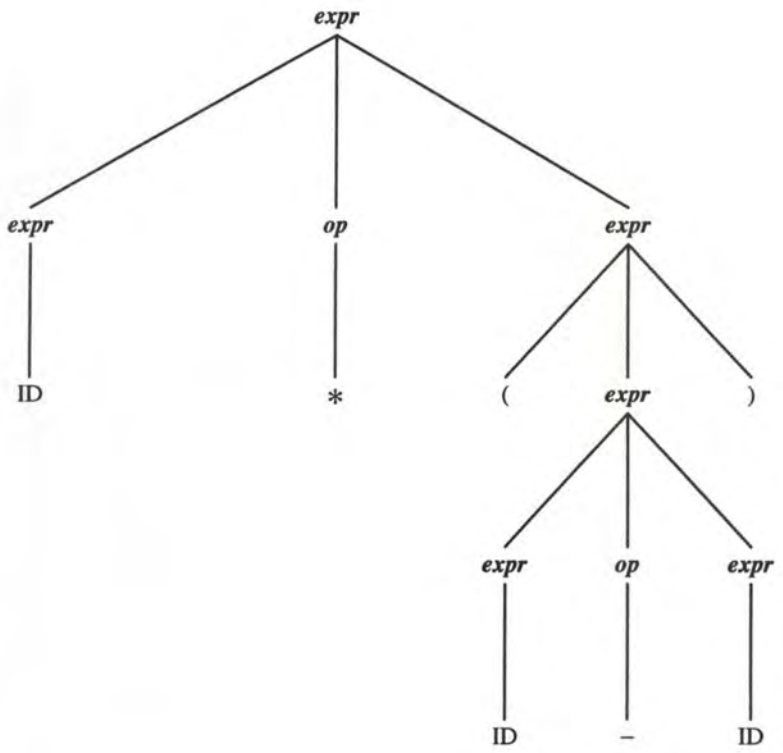
Nous avons fait remarquer à la page précédente qu'il y avait une analogie entre la dérivation d'une chaîne à partir de l'axiome d'une grammaire et le processus de résolution d'un programme Prolog. Au début du second chapitre on avait mis en évidence le fait que les choix atomes/clauses effectués à chaque étape de la résolution n'étaient pas neutres quant aux résultats de celle-ci. De la même manière, les choix concernant les remplacements de symboles non-terminaux dans la dérivation d'une chaîne ne sont pas indépendants du résultat. Montrons le sur un exemple. Soit la grammaire :

expr → ***expr op expr*** | (***expr***) | −***expr*** | ID
op → + | − | * | /

Nous allons donner deux dérivations différentes pour une même chaîne appartenant au langage défini par cette grammaire.

expr	***expr***
expr op expr	***expr op expr***
expr * ***expr***	ID ***op expr***
expr * (***expr***)	ID * ***expr***
expr * (***expr op expr***)	ID * (***expr***)
expr * (***expr*** − ***expr***)	ID * (***expr op expr***)
ID * (***expr*** − ***expr***)	ID * (ID ***op expr***)
ID * (ID − ***expr***)	ID * (ID − ***expr***)
ID * (ID − ID)	ID * (ID − ID)

On constate que si la chaîne dérivée est fort heureusement la même, plusieurs processus de dérivation différents permettent d'y arriver. On remarque également que les arbres syntaxiques font disparaître l'ordre dans lequel les substitutions ont été effectuées. Comme on peut le voir à la page suivante, la représentation par arbre syntaxique des deux dérivations données est la même.



Il est possible dans certains cas de rendre univoque le processus de dérivation pour une grammaire donnée en utilisant des conventions[4]. On peut par exemple toujours remplacer en premier le symbole le plus à gauche (ou le plus à droite) de la chaîne développée. On utilise alors le terme de dérivation gauche (droite) pour désigner cette manière de dériver une chaîne à partir du symbole distingué. La deuxième dérivation (celle de droite) de notre exemple de la page précédente est une dérivation gauche pour la chaîne ID*(ID–ID).

A.4. Ambiguîté des grammaires non contextuelles.

Il est malheureusement des cas où une quelconque convention de dérivation quant au choix du symbole non-terminal suivant à remplacer ne permet pas de définir de manière univoque le processus de dérivation. Typiquement, cela se produit lorsque, étant donné une chaîne $\alpha X \beta$ (avec α et β des chaînes de symboles et X un symbole non-terminal), il existe deux productions définissant X qui permettent de dériver la chaîne terminale voulue. Voici un exemple :

concat*→*liste liste
liste*→*liste elem* | *elem
elem→A | B | C | ε

[4]En toute généralité, cela n'est possible que si la grammaire n'est pas ambigüe

Etant donné une telle grammaire il est évident qu'il existe plus d'une dérivation possible pour la chaîne ABCAB par exemple. De manière générale il n'existe aucune dérivation univoque pour toute chaîne autre que la chaîne vide.

Si l'on pense en terme d'arbres syntaxiques on constate que de telles grammaires admettent plus d'un arbre syntaxique pour une même chaîne. De telles grammaires sont dites ambiguës. Il faut dès lors être prudent lors de l'écriture d'une grammaire non-contextuelle. Etant donné la remarque que nous venons de faire il est facile de montrer qu'une grammaire est ambiguë. Il suffit pour cela de donner un exemple de chaîne admettant plus d'une dérivation. Lorsque nous parlerons des grammaires attribuées nous verrons que celles-ci permettent de définir une sémantique[5] pour les chaînes valides du langage défini. Avec une grammaire attribuée ambiguë on peut de ce fait avoir plusieurs sémantiques différentes pour une même chaîne ce qui est souvent dangereux. Il est parfois possible de lever les ambiguïtés à l'aide de conventions. Par exemple, dans le cas des expressions arithmétiques en notation infixée les parenthèses et les priorités des opérateurs permettent d'avoir une sémantique unique pour toute chaîne valide.

Après nous être intéressés aux grammaires non-contextuelles nous allons maintenant parler des grammaires attribuées, montrer en quoi elles se ressemblent et ce que l'intégration d'attributs dans une grammaire peut nous apporter.

B. Grammaires attribuées.

Nous l'avons déjà dit les grammaires attribuées permettent de définir la sémantique d'un langage. Il existe deux types de notations pour ce faire : les définitions dirigées par la syntaxe et les schémas de traduction. La différence principale entre ces deux types de grammaires consiste en leur niveau d'abstraction. Les définitions dirigées par la syntaxe constituent des spécifications de haut niveau. Les règles sémantiques y sont souvent définies sous forme d'équations entre attributs ou sous toute autre forme équivalente statique. Ce caractère statique des règles sémantiques permet en particulier une totale indépendance quant à l'ordre de parcours de l'arbre syntaxique pour l'évaluation des valeurs d'attributs. Les définitions dirigées par la syntaxe cachent généralement la plupart des détails d'implémentation. C'est ce type de grammaire qui nous intéressera exclusivement au prochain chapitre.

Le second type de grammaire attribuée que constituent les schémas de traduction permet entre autres de faire apparaître certains détails d'implémentation. Par exemple, il sera fréquent d'effectuer des opérations générant des effets de bord. L'ordre d'évaluation des attributs ne sera dès lors généralement pas indépendant du résultat.

[5]Ce n'est pas obligatoirement toujours le cas. On peut imaginer d'autres utilisations des grammaires attribuées.

Nous allons maintenant entrer dans le vif du sujet en donnant une définition quelque peu formelle de ce qu'est un grammaire attribuée que nous ferons suivre par un exemple simple destiné à clarifier les choses.

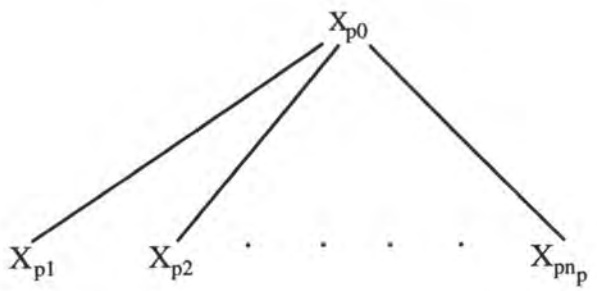
B.1. Définition.

Soit une grammaire non-contextuelle G=(S,N,A,P). On peut définir une sémantique pour cette grammaire en lui ajoutant un certain nombre de règles sémantiques. Concrètement, on ajoute à chaque symbole $X \in S$ un ensemble fini d'attributs noté A(X). A(X) est partitionné en deux sous-ensembles disjoints : les attributs synthétisés $A_0(X)$ et les attributs hérités $A_1(X)$[6]. Deux restrictions sont cependant applicables. Pour chaque symbole terminal T de la grammaire on a $A_0(T)=\emptyset$. De même pour le symbole distingué A on à $A_1(A)=\emptyset$. Chaque attribut α a un domaine de valeurs V_α. Supposons que l'ensemble P comporte m productions de la forme :

$$X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn_p}$$

avec $1 \leq p \leq m$ et $n_p \geq 0$. Les règles sémantiques sont en fait des fonctions $f_{pj\alpha}$ définies pour chaque attribut de chaque symbole apparaissant dans chaque production ($1 \leq p \leq m$, $0 \leq j \leq n_p$ et $\alpha \in A_0(X_{pj})$ si j=0 ou $\alpha \in A_1(X_{pj})$ si j>0). Chaque règle sémantique associée à une production $X_0 \rightarrow X_1 X_2 \ldots X_{n_p}$ définit une fonction de $V_{\alpha 1} \times V_{\alpha 2} \times \ldots \times V_{\alpha t}$ vers $V_{\alpha k}$ avec $\alpha_1, \alpha_2, \ldots, \alpha_t \in \bigcup_{i=0}^{n_p} A(X_i)$ et α_k un attribut d'un des symboles $X_0, X_1, X_2, \ldots, X_{n_p}$.

Les règles sémantiques d'une grammaire attribuée peuvent définir une sémantique pour les chaînes d'un langage de la manière suivante. Considérons l'arbre syntaxique représentant la dérivation d'une chaîne donnée. Si, à un certain moment on a appliqué la production $X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn_p}$ ($1 \leq p \leq m$), la partie correspondante de l'arbre syntaxique est :



Considérons un noeud X de cet arbre partiel et un de ses attributs α. On a $X=X_{p0}$ si $\alpha \in A_0(X)$ et $X=X_{pj}$ si $\alpha \in A_1(X)$, $1 \leq j \leq n_p$ car on considère une production unique et donc toute règle sémantique associée à cette production ne peut faire intervenir que des

[6]Nous verrons bientôt ce qu'il faut entendre par attribut synthétisé et attribut hérité.

attributs des symboles de celle-ci. La valeur v de l'attribut α au noeud X est donnée par la fonction $f_{pj\alpha} : V_{\alpha 1} \times V_{\alpha 2} \times \ldots \times V_{\alpha t} \rightarrow V_{\alpha}$ telle que :

$$v = f_{pj\alpha}(v_1, v_2, \ldots, v_t)$$

avec v_1, v_2, ..., v_t les valeurs des attributs $\alpha_1, \alpha_2, \ldots, \alpha_t$ aux noeuds X_{pk_1}, X_{pk_2}, ..., X_{pk_t} ($\{k_1, k_2, \ldots, k_t\} \subseteq \{0, 1, \ldots, n_p\}$). Le même processus de calcul de valeurs pour les attributs est appliqué à l'arbre syntaxique entier jusqu'à ce que l'ensemble des attributs des symboles étiquetant chacun des sommets de l'arbre ait été calculé. A l'issue de cette évaluation les valeurs des attributs de l'axiome (qui se trouve à la racine de l'arbre) définissent la sémantique de la chaîne dérivée. Ajoutons encore qu'un arbre syntaxique accompagné de la représentation des attributs et de leur valeur à chacun de ses sommets est appelé arbre décoré ou annoté.

Il est maintenant temps de donner un exemple de grammaire attribuée et de montrer comment s'effectue l'annotation d'un arbre syntaxique de celle-ci. Considérons la grammaire[7] suivante qui définit la syntaxe et la sémantique des expressions d'une calculette de bureau :

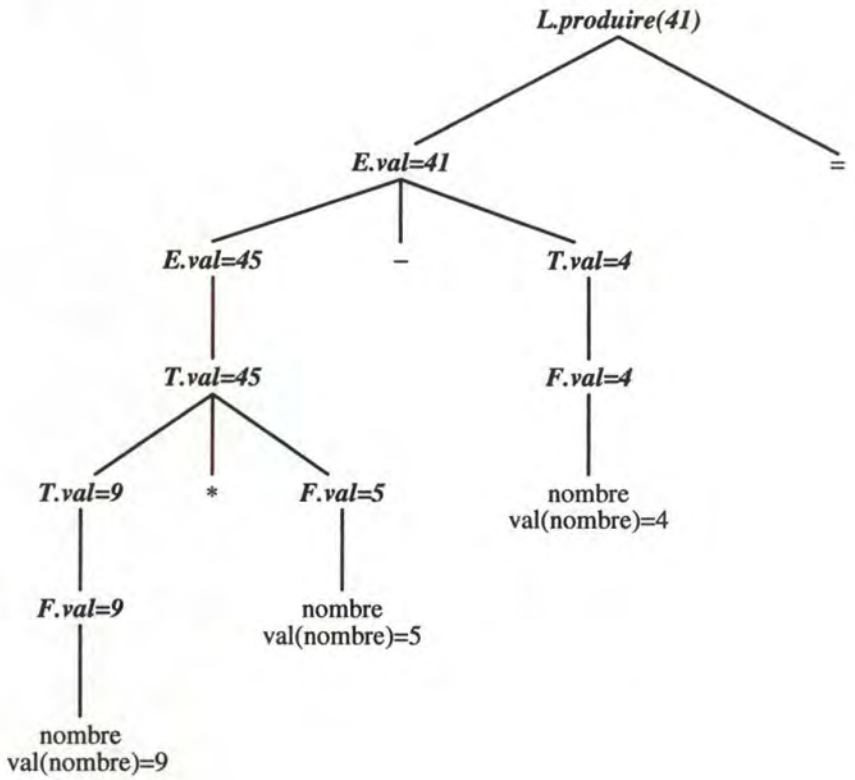
Productions	Règles sémantiques
$L \rightarrow E=$	Produire(E.val)
$E_0 \rightarrow E_1+T$	E_0.val:=E_1.val+T.val
$E_0 \rightarrow E_1-T$	E_0.val:=E_1.val−T.val
$E \rightarrow T$	E.val:=T.val
$T_0 \rightarrow T_1*F$	T_0.val:=T_1.val*F.val
$T_0 \rightarrow T_1/F$	T_0.val:=T_1.val/F.val
$T \rightarrow F$	T.val:=F.val
$F \rightarrow (E)$	F.val:=E.val
$F \rightarrow$nombre	F.val:=val(nombre)

On remarque que la grammaire n'utilise qu'un seul attribut "val" définissant le résultat et la manière dont on le calcule pour chaque type d'expression correspondant à un symbole non-terminal donné. La valeur de cet attribut val pour le symbole terminal "nombre" est déduite directement de la chaîne le représentant au moyen de l'opérateur "val(x)".

Si l'on regarde la première production et la règle sémantique associée on s'apperçoit que L n'a pas d'attribut à proprement parler. Il s'agit plutôt d'un attribut factice destiné à simuler l'affichage du résultat de l'évaluation de l'expression. On remarquera également la décomposition des expressions en sous-expressions de termes (symbole T) et de facteurs (symbole F). Une telle distinction permet d'assurer la priorité des opérateurs "*"

[7]Précisons que cette grammaire n'utilise que des attributs synthétisés.

et "/" sur "+" et "−". Pour compléter notre exemple nous donnons ci-dessous l'arbre syntaxique décoré pour l'expression 9*5−4=.



Nous avons déjà utilisé à plusieurs reprises les termes d'attribut synthétisé et d'attribut hérité sans jamais préciser de quoi il s'agit. Nous allons remédier à cette lacune.

B.2. Attributs synthétisés et attributs hérités.

Un attribut est dit synthétisé si sa valeur calculée pour un sommet de l'arbre syntaxique dépend uniquement des valeurs des (de certains) attributs des symboles associés aux sommets fils et/ou frères de ce sommet. Par exemple l'attribut "val" de notre grammaire "calculette" est un attribut synthétisé : sa valeur ne dépend que des valeurs des attributs "val" aux sommets fils de celui auquel s'effectue l'évaluation. On dit d'une grammaire n'utilisant que des attributs synthétisés qu'elle est S-attribuée. Une caractéristique intéressante de ce type de grammaire est qu'un arbre syntaxique peut toujours être annoté en effectuant un calcul de bas en haut. On commence par évaluer les attributs aux sommets feuilles et on évalue ensuite de manière incrémentale les attributs des autres sommets en remontant dans l'arbre vers la racine.

Un attribut hérité est un attribut calculé à partir des valeurs d'attributs aux sommets père et/ou frères du sommet auquel est attaché cet attribut. Même s'il est toujours possible de donner un équivalent S-attribué d'une grammaire utilisant des attributs hérités, les

sémantiques incorporant des attributs hérités sont souvent plus naturelles et plus facilement appréhendables.

A titre d'exemple de grammaire mixte nous empruntons à [4] la grammaire suivante définissant la sémantique des nombres binaires.

Productions	Règles sémantiques
$B \rightarrow 0$	$B.v=0$
$B \rightarrow 1$	$B.v=2^{s(B)}$
$L \rightarrow B$	$L.v=B.v$ $B.s=L.s$ $L.l=1$
$L_1 \rightarrow L_2 B$	$L_1.v=L_2.v+B.v$ $B.s=L_1.s$ $L_2.s=L_1.s+1$ $L_1.l=L_2.l+1$
$N \rightarrow L$	$N.v=L.v$ $L.s=0$
$N \rightarrow L_1.L_2$	$N.v=L_1.v+L_2.v$ $L_1.s=0$ $L_2.s=-L_2.l$

Explicitons la signification des différents attributs.

L'attribut "v" est un nombre rationnel donnant la valeur décimale de la chaîne binaire associée au symbole non-terminal de l'attribut considéré. L'attribut "s", un entier, désigne la puissance de 2 du bit le plus à droite dans une chaîne binaire. Il est utilisé en ce sens dans la seconde production. Quant à l'attribut "l" son utilité est de calculer la puissance de 2 du bit le plus à droite de la chaîne se trouvant à droite du point décimal. L'attribut "l" est un entier et est utilisé dans la dernière production.

Remarquez l'utilisation d'attributs synthétisés (B.v,L.v,L.l et N.v) et d'attributs hérités (B.s et L.s). L'avantage des attributs hérités est qu'ils permettent la manipulation d'informations liées à des dépendances contextuelles. Dans l'exemple précédent l'attribut "s" qui permet de connaître la puissance de 2 d'un bit est lié au contexte puisque sa valeur dépend de la longueur du suffixe droit de la chaîne dans laquelle il se trouve. On remarquera également le caractère un peu artificiel de l'attribut "l". On aurait par exemple pu calculer la valeur de la chaîne de bits à droite du point décimal de la même manière que pour celle de gauche c'est-à-dire en attribuant des puissances de 2 décroissantes aux différents bits de cette chaîne.

B.3. Cohérence des grammaires attribuées.

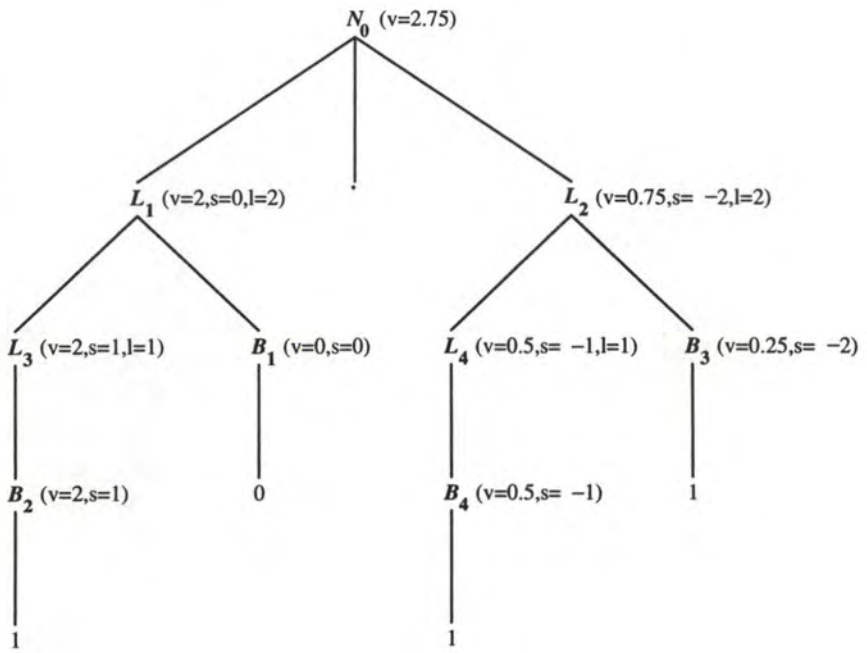
Nous avons déjà dit que la combinaison d'attributs synthétisés et hérités permettait le plus souvent d'obtenir une définition naturelle et claire de la sémantique d'un langage. Le danger de l'utilisation conjointe de ces deux types d'attributs est qu'elle peut donner lieu à des définitions circulaires c'est-à-dire pour lesquelles le processus d'évaluation d'attributs boucle indéfiniment. Pour des petites grammaires telles que celle présentée pour les nombres binaires, il est possible de se convaincre assez facilement que celle-ci est non circulaire. Pour des grammaires de grande taille (et c'est souvent le cas pour les langages de programmation courants) s'assurer de l'absence de définitions circulaires peut être beaucoup moins évident. Le calcul des attributs d'un arbre syntaxique n'impose aucun ordre de parcours particulier pour les sommets de celui-ci. La seule règle à respecter est que tout attribut doit être calculé après ceux dont il dépend. Il est bien sur toujours possible de n'utiliser que des attributs synthétisés. Dans pareil cas l'on est assuré que la grammaire correspondante est non circulaire et le parcours en profondeur d'abord des arbres syntaxiques engendrés constitue une solution acceptable. Il calculera toujours les attributs "fils" avant les attributs "pères". De plus, celui-ci possède les caractéristiques d'être efficace et facilement implémentable. Malheureusement, il n'est pas toujours souhaitable de se passer des attributs synthétisés. Il est dès lors utlile de disposer d'une méthode automatique pour tester la circularité d'une grammaire attribuée. Nous allons en dire un mot avant de clore ce chapitre consacré aux grammaire attribuées.

Les interdépendances entre les attributs des symboles peuvent être aisément représentées à l'aide d'un graphe orienté appelé graphe de dépendances. La structure d'un tel graphe est simple. Chaque attribut de chaque symbole de l'arbre est représenté par un sommet dans le graphe de dépendances. Si un attribut α dépend d'un attribut β, le graphe possède un arc allant du sommet associé à β vers celui associé à α. Nous donnons à titre d'exemple l'arbre syntaxique et le graphe de dépendances obtenus pour la dérivation de la chaîne binaire 10.11 (voir page suivante).

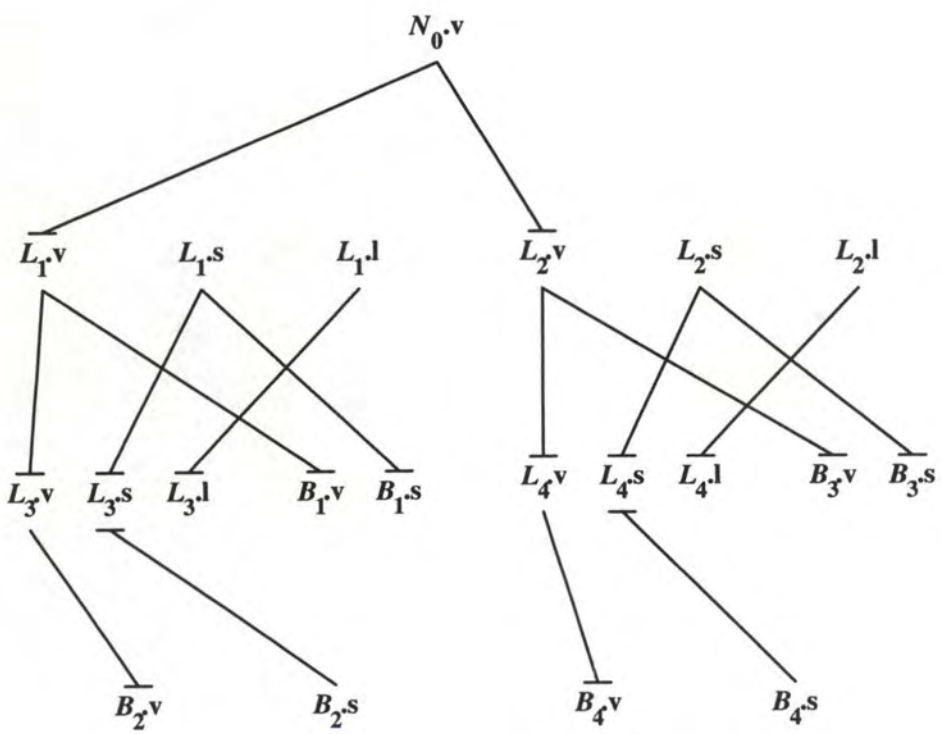
On peut déterminer si le calcul d'attributs pour la chaîne 10.11 donne lieu à une ou des références circulaires en observant le graphe de dépendances associé. Il suffit pour cela de regarder si celui-ci ne contient pas de cycles. En effet, un groupe d'attributs dont les sommets correspondants du graphe se trouvent sur un même cycle, dépendent chacun de l'attribut associé au sommet qui les précède sur le cycle. Dans le cas de notre exemple on peut facilement se convaincre que ce n'est pas le cas.

Cette représentation par graphe de dépendances peut également être utilisée pour déterminer un ordre de parcours valide pour les sommets (ou plus précisément les attributs des symboles utilisés) de l'arbre syntaxique.

Arbre de dérivation pour la chaîne 10.11



Graphe de dépendances pour l'arbre syntaxique de la chaîne 10.11



Un tri topologique d'un graphe orienté acyclique est un ordonnancement quelconque des sommets du graphe tel que, pour tous les arcs du graphe, le sommet de départ apparaisse dans l'ordonnancement avant le sommet d'arrivée. Par exemple, si i et j sont deux sommets du graphe et que (i,j) est un arc, alors i doit apparaître avant j dans l'ordre de parcours des sommets. Il est évident que tout tri topologique des sommets du graphe de dépendances associé à une dérivation donnée donne un ordre possible pour le calcul des attributs.

Si la méthode qui consiste à construire le graphe de dépendances pour chaque arbre syntaxique engendré et à tester l'existence d'un cycle dans celui-ci est correcte et utilisable, elle présente cependant l'inconvénient d'exiger la construction systématique d'un graphe de dépendances. Il serait plus beaucoup plus commode et pratique de disposer d'un test directement applicable à la grammaire étudiée permettant de dire si celle-ci est circulaire ou non. Une telle méthode et un algorithme dérivé existent. Nous ne présenterons pas cet algorithme intégralement, nous nous contenterons d'énoncer son principe de fonctionnement. Le lecteur intéressé trouvera une description complète et formelle de celui-ci dans [4]. L'idée de base de cet algorithme s'énonce ainsi. Soit une production P : $X_0 \rightarrow X_1 X_2 ... X_n$. On construit le graphe de dépendances D_p de P comme précédemment c'est-à-dire en restreignant l'ensemble des sommets aux attributs des symboles $X_0, X_1, X_2, ... X_n$. Soit D_j un graphe orienté acyclique associé à X_j ($1 \leq j \leq n$). Chaque arc de D_j est momentanément ajouté à D_p. Si le graphe résultant contient un cycle alors la grammaire est circulaire. Dans le cas contraire les chemins de ce graphe constituent alors un nouveau graphe orienté acyclique qui est ajouté à un ensemble $F(X_0)$. Le même processus se répète jusqu'à ce que les ensembles F(X) (avec X les non-terminaux de la grammaire) soient complètement définis ou que la grammaire soit prouvée circulaire.

Au cours de ces trois premiers chapitres nous avons successivement parlé de logique de prédicats du premier ordre, de programmation logique en Prolog, d'interprétation abstraite et de grammaires attribuées. Il est maintenant temps de mettre tout cela ensemble et de présenter ce qu'est l'interprétation abstraite par grammaires attribuées pour programmes Prolog.

CHAPITRE IV : INTERPRETATION ABSTRAITE ET GRAMMAIRES ATTRIBUEES

Chapitre IV : interprétation abstraite et grammaires attribuées.

La méthode d'interprétation abstraite présentée dans ce chapitre a été à l'origine décrite par K. Musumbu et K. Barbar dans [2]. Les idées qui sous-tendent cette méthode sont les suivantes. Le domaine abstrait consiste en un ensemble de substitutions abstraites dont l'objet est de contenir des informations utiles sur les variables d'un programme Prolog. Pour ce domaine abstrait on définit une série d'opérateurs abstraits qui seront ensuite utilisés pour le calcul des attributs de la grammaire. Ainsi construite, la grammaire attribuée matérialise une sémantique "abstraite" de la transformation de fonctions associée à tout programme Prolog. Le tout permettant en quelque sorte de faire de l'interprétation abstraite dirigée par la syntaxe. Etant donné un programme P et un but $p(t_1,...,t_n)$ avec p un prédicat défini dans P et $t_1,...,t_n$ des termes, le but est d'obtenir des informations "a priori" sur le comportement du programme P pour le but donné.

Dans ce qui va suivre nous procèderons de manière semblable à ce que nous avons fait au second chapitre. Nous présenterons d'abord les caractéristiques obligatoires pour tout domaine abstrait utilisable avec la méthode d'interprétation abstraite par grammaire attribuée. Un exemple de domaine abstrait simple sera ensuite présenté ainsi que les opérateurs abstraits associés. Le troisième point abordé sera la grammaire en elle-même. Nous la donnerons sous forme de grammaire non contextuelle dans un premier temps. Les règles sémantiques seront décrites immédiatement après. La dernière partie du chapitre sera consacrée à l'algorithme de calcul de point fixe dans lequel nous retrouverons bon nombre des idées énoncées à la fin du chapitre 2.

A. Domaines abstraits.

A.1. Fondements des domaines abstraits utilisables.

Considérons un programme Prolog quelconque P et l'ensemble (fini) des variables utilisées dans celui-ci, soit $D=\{x_1,...,x_n\}$. Comme nous l'avons vu au premier chapitre, un tel programme muni d'un but fournira une réponse qui sera "oui" ou "non" avec en plus généralement une substitution $\{y_1/t_1,...,y_k/t_k\}$ avec $\{y_1,...,y_k\}$ l'ensemble des variables apparaissant dans le but donné au programme et $t_1,...,t_k$ des termes. L'ensemble des substitutions formées à partir de tels termes et des variables du programme est noté PS_D. L'ensemble des parties de PS_D ($P(PS_D)$) constituera l'ensemble concret de notre méthode d'interprétation abstraite. Cet ensemble sera noté CS_D ($CS_D=P(PS_D)$). Le domaine abstrait correspondant sera construit de la manière suivante. L'interpréteur abstrait étudiera le comportement d'un programme donné, non plus à partir de substitutions normales, mais bien de substitutions abstraites. Celles-ci seront de la forme $\{x_1/v_1,...,x_n/v_n\}$. $x_1,...,x_n$ étant des variables du programme étudié et $v_1,...,v_n$ des "valeurs" appartenant à un ensemble T_A donné. Bien que les éléments de cet

ensemble puissent être de nature quelconque, l'ensemble des substitutions abstraites associées au programme P, que nous noterons AS_D, devra constituer un treillis complet. Aussi certaines contraintes et caractéristiques seront requises.

En premier lieu il sera souhaitable que les éléments de T_A véhiculent des informations utiles et cohérentes concernant les variables (ou autres objets manipulés par le programme) auxquels ils sont associés dans toute substitution abstraite.

A la fin du chapitre consacré à l'interprétation abstraite, nous avions évoqué la possibilité que l'algorithme de calcul de point fixe "tourne" indéfiniment si le domaine abstrait n'est pas fini. Dans notre cas le domaine abstrait AS_D ne peut être un ensemble fini que si T_A, l'ensemble des "valeurs abstraites" est également fini. On utilisera donc de préférence des ensembles T_A finis.

Il vient d'être rappelé que AS_D doit constituer un treillis complet. On supposera donc l'existence d'une relation d'ordre partiel[1] sur les substitutions abstraites (les éléments de AS_D) et cela pour n'importe quel programme Prolog.

L'existence d'un ordre partiel sur les éléments du domaine abstrait ne suffit pas pour que celui-ci constitue un treillis complet (cfr définition d'un treillis complet au chap. 2). Il faut également que tout sous-ensemble de AS_D possède une plus petite borne supérieure et une plus grande borne inférieure dans AS_D. Pour ce faire tout domaine abstrait utilisé possèdera donc deux éléments notés $\bot$ et $\top$, correspondant respectivement à $glb(AS_D)$ et à $lub(AS_D)$[2].

AS_D étant un treillis complet, il reste à assurer la cohérence des approximations effectuées par ses éléments par rapport aux substitutions standard. Il suffit pour cela de définir les fonctions d'abstraction et de concrétisation, respectivement α et γ. Rappelons que les contraintes à respecter pour ces deux fonctions sont :

$$\forall\, s \in CS_D : \{s\} \subseteq \gamma(\alpha(s))$$
$$\forall\, s \in AS_D : \alpha(\gamma(s)) = s$$

Remarquez que la fonction de concrétisation γ sert également à définir la sémantique des substitutions abstraites de AS_D.

Ayant montré comment est construit un domaine abstrait pour la méthode d'interprétation abstraite qui nous occupe, nous pouvons en donner un exemple.

[1]Nous ne définissons pas cette relation pour le moment car celle-ci est généralement dépendante de T_A.
[2]En toute généralité $lub(AS_D)$ ne sera pas toujours nécessaire.

A.2 Domaine abstrait : un exemple.

Le domaine abstrait que nous allons donner est généralement appelé domaine des modes. Chaque élément de l'ensemble des "valeurs abstraites" donne une indication de l'état dans lequel se trouve la variable à laquelle il est associé dans une substitution abstraite. L'ensemble T_A pour le domaine abstrait de mode est :

$$\{ground,var,any\}$$

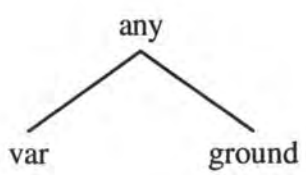
De manière informelle la sémantique des éléments du type mode peut être décrite comme suit : "ground" signifie que la variable qu'il qualifie est liée à un terme clos. "var" décrit une variable libre (non encore liée). "any" quant à lui, représente un terme qui est soit un terme clos soit une variable mais sans qu'il soit possible de déterminer sa nature exacte. "any" regroupe également tous les termes non clos c'est-à-dire de la forme $f(t_1,...,t_n)$ et qui contiennent au moins une variable.

Les éléments de AS_D (pour un ensemble de variables de programme D), les substitutions abstraites, sont donc de la forme :

$$\{v_1/a_1,v_2/a_2,...,v_n/a_n\}\ (n>0)$$

avec $v_1,v_2,...,v_n$ des variables appartenant à D et $a_1,a_2,...,a_n$ des éléments de T_A[3].

On définit une relation d'ordre partiel $\leq$ sur les éléments de T_A. Celle-ci est représentée par la figure ci-dessous.



On remarque que var et ground ne sont pas comparables. Il s'agit donc bien d'un ordre partiel.

La relation $\leq$ étant définie pour les "valeurs abstraites" nous pouvons donner un relation d'ordre partiel pour les substitutions abstraites que nous noterons également par $\leq$. Celle-ci est définie par :

$$\beta\leq\beta' \Leftrightarrow \forall\ x\in dom(\beta)\cup dom(\beta') : x\beta\leq x\beta'$$

[3]Par la suite nous utiliserons la lettre grecque β pour symboliser les substitutions abstraites.

avec β et β' des substitutions abstraites et dom(β) et dom(β') l'ensemble des variables apparaissant dans β et β' respectivement.

On peut se convaincre assez aisément que, muni d'une telle relation d'ordre, AS_D constitue bien un treillis complet. Par exemple la plus petite borne supérieure d'un sous-ensemble $X=\{\beta_1,...,\beta_m\}$ de AS_D sera donnée par lub(X) lequel est défini comme :

$$lub(\{\beta_1,...,\beta_i\})=lub(\{lub(\{\beta_1,...,\beta_{i-1}\}),\beta_i\}) \quad \text{si } i>2$$
$$lub(\beta,\beta')=\{x_1/a_1,x_2/a_2,...,x_n/a_n\} \quad \forall\, \beta,\beta' \in AS_D$$

avec :

$$\{x_1,x_2,...,x_n\}=dom(\beta)\cup dom(\beta') \quad \text{et}$$
$$a_i=lub(x_i\beta,x_i\beta') \quad \text{si } x_i \in dom(\beta) \text{ et } x_i \in dom(\beta')$$
$$=x_i\beta \quad \text{si } x_i \in dom(\beta)$$
$$=x_i\beta' \quad \text{si } x_i \in dom(\beta')$$

et l'opérateur $lub(a_1,a_2)$ défini par la table suivante :

a_1 \ a_2	var	ground	any	$\perp$
var	var	any	any	var
ground	any	ground	any	ground
any	any	any	any	any
$\perp$	var	ground	any	$\perp$

Avant de pouvoir définir les opérateurs abstraits adaptés au domaine que nous venons de construire, il est nécessaire de dire un mot concernant la sémantique qu'associe à tout programme Prolog la grammaire attribuée que nous allons utiliser.

Soit un programme Prolog P et D l'ensemble des variables de ce programme. La sémantique abstraite de P est définie par un ensemble de triplets $(\beta_{in},p,\beta_{out})$ avec $p \in$ Pred(P) l'ensemble des symboles de prédicats définis dans P et β_{in},β_{out} des substitutions abstraites appartenant à AS_D. On définit la fonction totale monotone eta comme :

$$eta : AS_D \times Pred(P) \rightarrow AS_D$$

Etant totale et monotone eta satisfait les conditions suivantes :

- $\forall\, (\beta_{in},p) \in AS_D \times Pred(P),\ \exists!\ \beta_{out} \in AS_D : \beta_{out}=eta(\beta_{in},p)$
- $\forall\, (\beta_1,p),(\beta_2,p) \in AS_D \times Pred(P),\ \beta_1 \leq \beta_2 \Rightarrow eta(\beta_1,p) \leq eta(\beta_2,p)$

Pour l'algorithme de calcul de point fixe, à tout moment sera associé au prédicat p une table de triplets $(\beta_{in},p,\beta_{out})$ avec $eta(\beta_{in},p)=\beta_{out}$. De manière informelle l'information

représentée par ces triplets est la suivante : si un programme P est lancé avec un but $p(x_1,x_2,...,x_n)$ et que β_{in} "décrit" l'état des variables $x_1,x_2,...,x_n$ à l'entrée du programme alors β_{out} constitue une approximation abstraite du résultat fourni par celui-ci pour le but donné. Comme on le verra par la suite la table eta d'un prédicat sert de support tant au calcul de point fixe qu'à la technique de memo-ization déja évoquée au second chapitre.

B. Opérateurs abstraits.

Dans ce qui suit nous allons définir des opérateurs abstraits pour le domaine abstrait type mode. Les valeurs abstraites "any", '"var" et "ground" seront respectivement notées "a", "v" et "g". Les mêmes opérateurs (avec peut-être d'autres en plus) sont nécessaires pour d'autres domaines abstraits. Les opérateurs donnés ci-dessous sont de deux types. Les premiers servent essentiellement à calculer des substitutions abstraites à l'entrée et à la sortie des noeuds des arbres syntaxiques engendrés par la grammaire pour le calcul de point fixe. Il s'agit des opérateurs $Restr_c$, $Restr_b$, Map, Ext_b, Abi_1 et Abi_2. Le second type d'opérateurs concerne avant tout la gestion des tables eta et la memo-ization. Ces opérateurs sont Ext_p et Adj. Voici maintenant les définitions formelles de ces opérateurs.

- Application d'une substitution abstraite à une variable :

$$x\beta = \begin{cases} \omega & \text{si } \exists\, x/\omega \in \beta \\ v & \text{sinon} \end{cases}$$

Ex: $x_2\{x_1/v,x_2/g\}=g$

Cet opérateur sert essentiellement de support aux autres opérateurs travaillant sur les substitutions abstraites.

- Restriction d'une substitution abstraite sur une autre :

$$Restr_c(\beta,\beta')=\{x_i/x_i\beta \mid x_i \text{ est une variable apparaissant dans } \beta'\}$$

Ex: $Restr_c(\{x_1/v,x_2/g\},\{x_1/g,x_2/a,x_3/v\})=\{x_1/v,x_2/g,x_3/v\}$

Opérateur utilisé pour calculer la substitution abstraite en sortie d'une clause.

- Application d'une substitution abstraite à une liste de variables :

$$Restr_b(\beta,(x_{i_1},...,x_{i_n}))=\{x_{i_1}/x_{i_1}\beta,...,x_{i_n}/x_{i_n}\beta\}$$

Ex: $Restr_b(\{x_1/v,x_2/a,x_3/a\},(x_1,x_2,x_3,x_6))=\{x_1/v,x_2/a,x_3/a,x_6/v\}$

Fonction semblable à celle de $Restr_c$. On utilise $Restr_b$ pour calculer une substitution abstraite en entrée pour un but.

- Mapping d'un ensemble de variables vers un autre pour les appels de prédicats :

$$Map(((x_{j_1},...,x_{j_n}),\beta,(x_{k_1},...,x_{k_n}))=\{x_{k_1}/x_{j_1}\beta,...,x_{k_n}/x_{j_n}\beta\}$$

Ex: $Map((x_2,x_4,x_5,x_7),\{x_2/v,x_7/g\},(x_2,x_3,x_4,x_7))=\{x_2/v,x_3/v,x_4/v,x_7/g\}$

Cet opérateur est utilisé pour créer les substitutions abstraites en entrée et en sortie d'un appel de prédicat.

- Extension de la substitution abstraite courante après un appel de prédicat :

$$Ext_b((x_{i_1},...,x_{i_n}),\beta_1,\beta_2)=\alpha(\theta')$$

Avec : $\theta' \in CS_D$ telle que :

$\exists\ \theta \in \gamma(\beta_1)$ et
$\exists\ \sigma \in CS_D$ une subsitution de sortie pour un but B telle que
$dom(\sigma)=(x_{i_1},...,x_{i_n})$ et
$\{x_{i_1},...,x_{i_n}\} \in dom(\gamma(\beta_2))$ et
$\theta'=\theta\sigma$

Ex:
$Ext_b((x_1,x_3,x_4),\{x_1/v,x_2/g,x_3/g,x_6/g,x_7/a\},\{x_1/a,x_3/g,x_7/v\})=\{x_1/a,x_2/g,x_3/g,x_6/g,x_7/v\}$

Cet opérateur est utilisé pour étendre à la substitution abstraite courante dans une suite de buts d'une même clause, la substitution abstraite en sortie pour un but de cette suite.

- Ajout d'un élément dans une table eta :

$Ext_p(eta,(\beta,p))=eta$ si $\exists\ (\beta,p,\beta') \in eta$
$=eta \cup (\beta,p,\perp)$ sinon

Est utilisé pour ajouter un triplet dans une table eta dont le domaine ne contient pas encore (β,p).

- Mise à jour d'une table eta :

$Adj((\beta_{in},p,\beta_{out}),eta)=eta$ si $\beta_{out} \leq eta(\beta_{in},p)$

$=eta\setminus[(\beta_{in},p,eta(\beta_{in},p))\cup(\beta_{in},p,\beta_{out})]$ sinon

La mise à jour d'une table eta est effectuée lorsque la substitution abstraite en sortie d'un appel de prédicat p avec une substitution d'entrée β_{in} constitue une approximation plus correcte du résultat réel que celle se trouvant déjà dans la table.

• Opérateur abstrait Abi_1 pour le built-in entre deux variables x et y. Fournit un équivalent abstrait de l'opération "x=y".

$Abi_1(x,y)$	ground	var	any
ground	g	g	g
var	g	v	a
any	g	a	a

• Opérateur abstrait Abi_2 pour le built-in entre une variable x et un foncteur f. Equivalent abstrait de "x=f(...)".

$$Abi_2(x=f(x_1,...,x_n),\beta)=\{x/\omega,x_1/\omega_1,...,x_n/\omega_n\}$$

Avec ω et ω_i (i=1..n) tels que :

$\omega=g \Leftrightarrow \omega_i=g \quad \forall\, i$
$\omega=a \Leftrightarrow \exists\, j : \omega_j \neq g$
$\forall\, j : mode(x_j)=Abi_1(x,x_j)$

C. La grammaire attribuée.

La grammaire que nous allons utiliser va nous permettre de définir la sémantique de la transformation de fonctions associée à tout programme Prolog. A tout prédicat sera associée une fonction eta dont la table est composée de triplets $(\beta_{in},p,\beta_{out})$ avec p le prédicat considéré, β_{in} la substitution abstraite en entrée et β_{out} la substitution abstraite résultant de l'appel du prédicat p avec β_{in} comme état initial. Etant donné un programme P et un couple (β_{in},p) la sémantique correspondante sera donnée par $eta(\beta_{in},p)$. Calculer la table de la fonction eta des différents prédicats d'un programme sera la tâche de notre grammaire attribuée. A chaque noeud de l'arbre syntaxique engendré pour un prédicat donné seront attachés deux attributs (β_{in} et β_{out}) représentant les substitutions abstraites en entrée et en sortie à ce noeud. La racine[4] de l'arbre comportera en plus un attribut eta

[4]Plus précisément chaque noeud de l'arbre syntaxique possède deux attributs eta_{in} et eta_{out} qui représentent respectivement la table de la fonction eta à l'entrée et à la sortie du noeud considéré

définissant l'état actuel de la fonction eta associée au prédicat concerné. C'est cette table eta du noeud racine qui nous intéressera principalement. A chaque itération de l'algorithme de calcul de point fixe, on effectuera un parcours de l'arbre syntaxique. A chaque étape correspondra une fonction eta définie par la table eta de la racine. La suite des fonctions correspondantes ainsi construite servira de base pour le calcul du plus petit point fixe. Nous retrouvons ici le principe présenté au second chapitre : le plus petit point fixe comme limite d'une suite monotone de fonctions.

La grammaire attribuée qui va suivre est du type schéma de traduction, les résultats étant dépendants de l'ordre de parcours des sommets de l'arbre syntaxique. Celle-ci est comme vous vous en doutez non circulaire. On remarquera son caractère mixte. Des attributs synthétisés et hérités étant utilisés. Malgré une telle caractéristique la grammaire présente néanmoins l'avantage de permettre le calcul des attributs en effectuant un parcours en profondeur d'abord sur les sommets de l'arbre syntaxique. Nous l'avons déjà dit cette méthode est facilement implémentable et efficace. Nous donnons ici un algorithme pour ce type de parcours.

```
procédure Parcours(n : Noeud);
  début
    pour tout sommet m fils de n, de gauche à droite faire
      Parcours(m);
  fin
```

C.1. Définition de la syntaxe.

Voici maintenant la grammaire qui sera utilisée. Nous la donnons dans un premier temps sous forme de grammaire non-contextuelle.

- Grammaire : G=(S,N,ROOT,P)

- Ensemble des symboles : S={ROOT,φ,SC,C,SB,B,$p(x_1,...,x_n)$,$p(x_{i_1},...,x_{i_n})$, $x_i=x_j$,$x_i=f(x_{i_1},...,x_{i_k})$} avec p un symbole de prédicat, $x_1,...,x_n$, x_i, x_j, $x_{i_1},...,x_{i_n}$ et $x_{i_1},...,x_{i_k}$ des variables et f un symbole de foncteur.

- Ensemble des productions : P composé de :

  p1 : $\boldsymbol{ROOT} \rightarrow p(x_1,...,x_n)\ \varphi$
  p2 : $\varphi \rightarrow \boldsymbol{SC}$
  p3 : $\varphi \rightarrow \varepsilon$
  p4 : $\boldsymbol{SC} \rightarrow \boldsymbol{C}$
  p5 : $\boldsymbol{SC} \rightarrow \boldsymbol{C}\ \boldsymbol{SC}$
  p6 : $\boldsymbol{C} \rightarrow \boldsymbol{SB}$
  p7 : $\boldsymbol{SB} \rightarrow \varepsilon$

p8 : $\boldsymbol{SB \rightarrow B\ SB}$
p9 : $\boldsymbol{B} \rightarrow x_i = x_j$
p10 : $\boldsymbol{B} \rightarrow x_i = f(x_{i_1},...,x_{i_k})$
p11 : $\boldsymbol{B} \rightarrow p(x_{i_1},...,x_{i_n})\ \varphi$

Avant de donner les règles sémantiques, il est nécessaire de faire deux remarques concernant la cette grammaire.

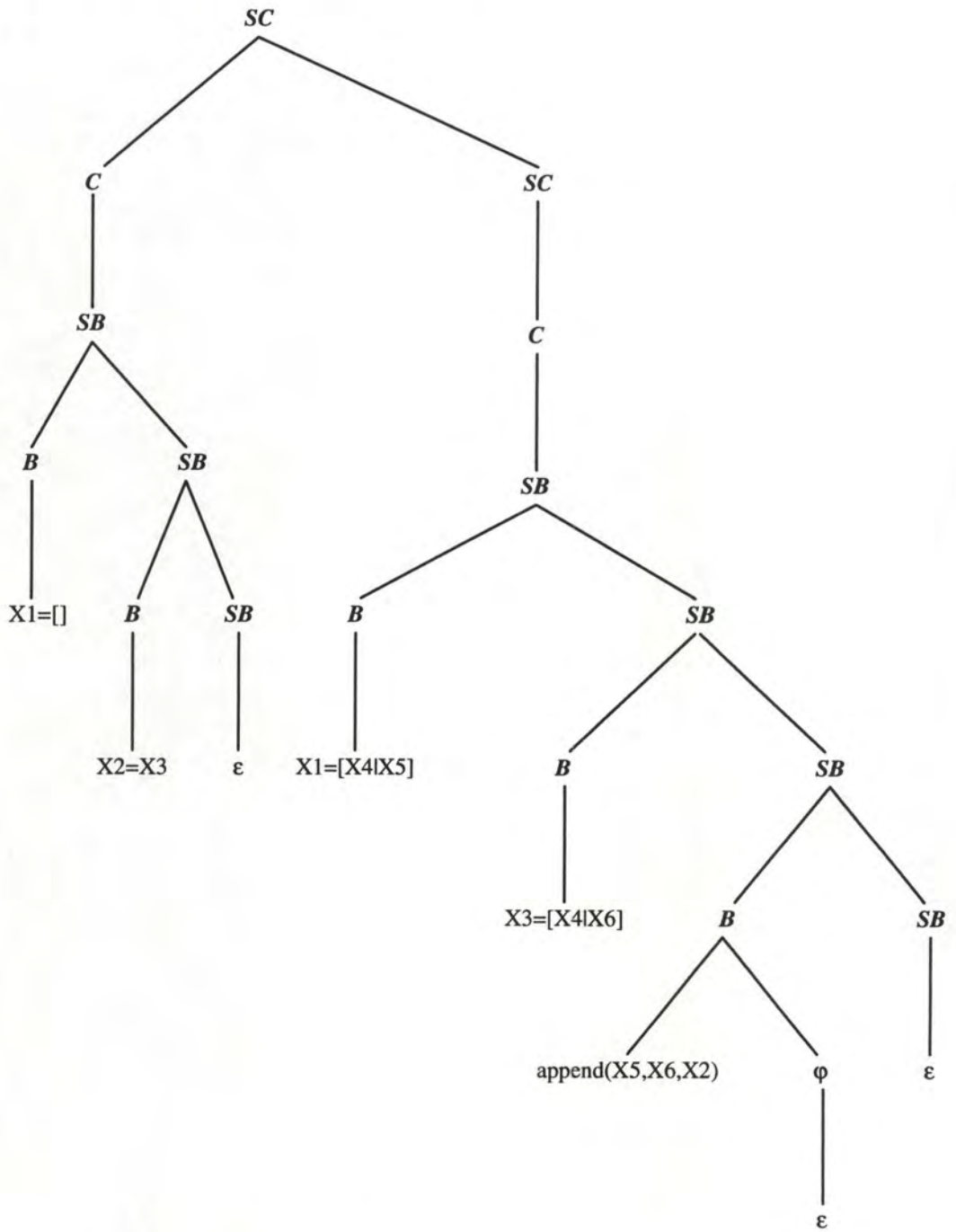
- Afin de rendre la grammaire plus concise nous n'avons pas défini les symboles de prédicats, de foncteurs ainsi que les variables. Ajouter des symboles non-terminaux définissant ceux-ci ne serait d'ailleurs pas vraiment utile pour présenter notre méthode d'interprétation abstraite, les règles sémantiques associées étant triviales. Les symboles p, f, $x_1,...,x_n$, x_i, x_j, $x_{i_1},...,x_{i_n}$, $x_{i_1},...,x_{i_k}$ ne sont donc pas des symboles terminaux.

- La grammaire ci-dessus ne permet pas de dériver la chaine de caractères que constitue tout programme Prolog. Son utilité est autre. Il s'agit de construire les arbres syntaxiques nécessaires au calcul de point fixe. La construction d'un arbre syntaxique pour un prédicat est effectuée à l'aide d'un schéma de traduction dont nous parlerons au chapitre suivant. Afin de clarifier les choses nous donnons un exemple de l'arbre construit pour un prédicat donné.

Soit le prédicat append défini par :

```
append(X1,X2,X3):-X1=[],X2=X3.
append(X1,X2,X3):-X1=[X4|X5],X3=[X4|X6],append(X5,X6,X2).
```

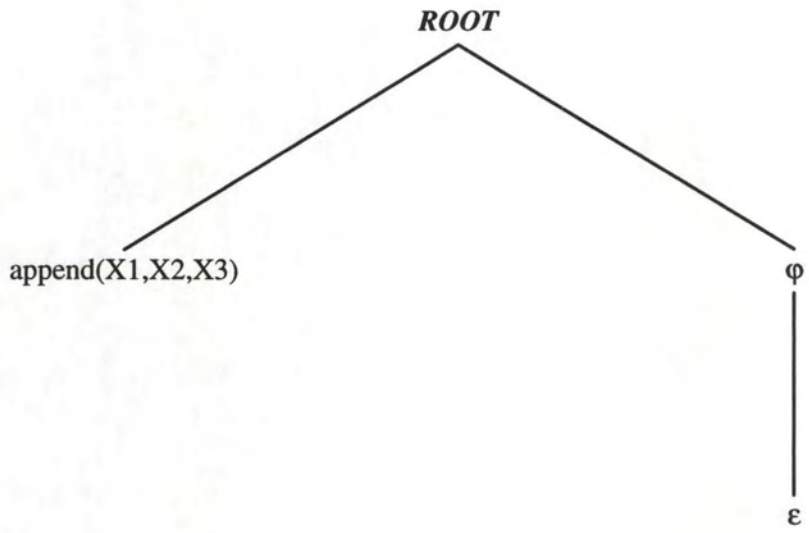
L'arbre syntaxique T_{append} engendré par la grammaire est :



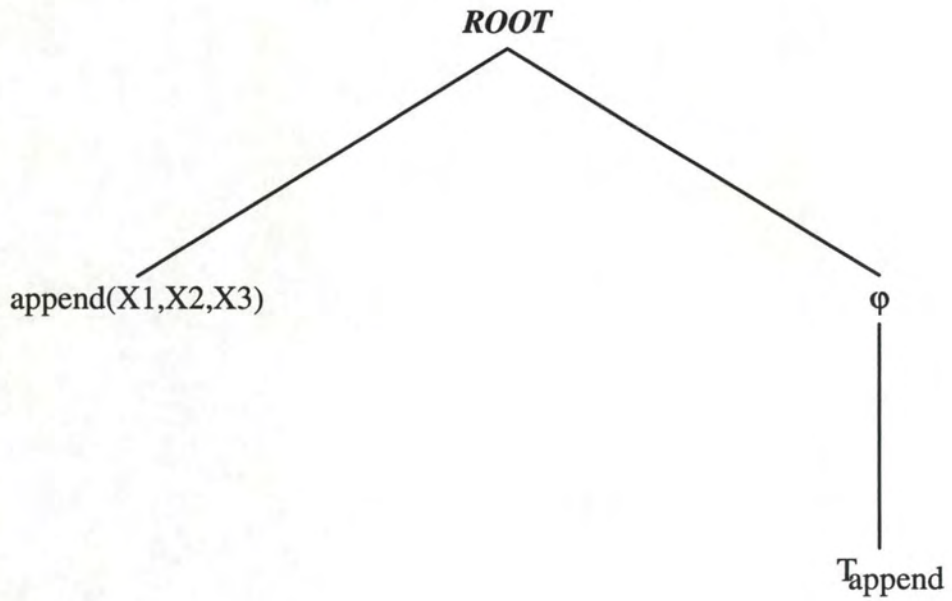
Attardons nous un instant à expliquer l'utilité du symbole φ. Lors de chaque itération du calcul de point fixe, il s'agit d'affiner les approximations des résultats intermédiaires utilisés lors de l'itération précédente. Dans notre cas on effectuera cela en étendant l'arbre syntaxique. Le noeud ε sera remplacé par l'arbre syntaxique du prédicat appelé au noeud φ (dans notre exemple il s'agit de append). Ainsi, à chaque itération, on étendra l'arbre en lui ajoutant un le ou les arbres syntaxiques des prédicats appelés aux noeuds

feuilles. Dans l'exemple donné la suite[5] $T_1,T_2,T_3,...$ des arbres engendrés sera la suivante.

T_1 :

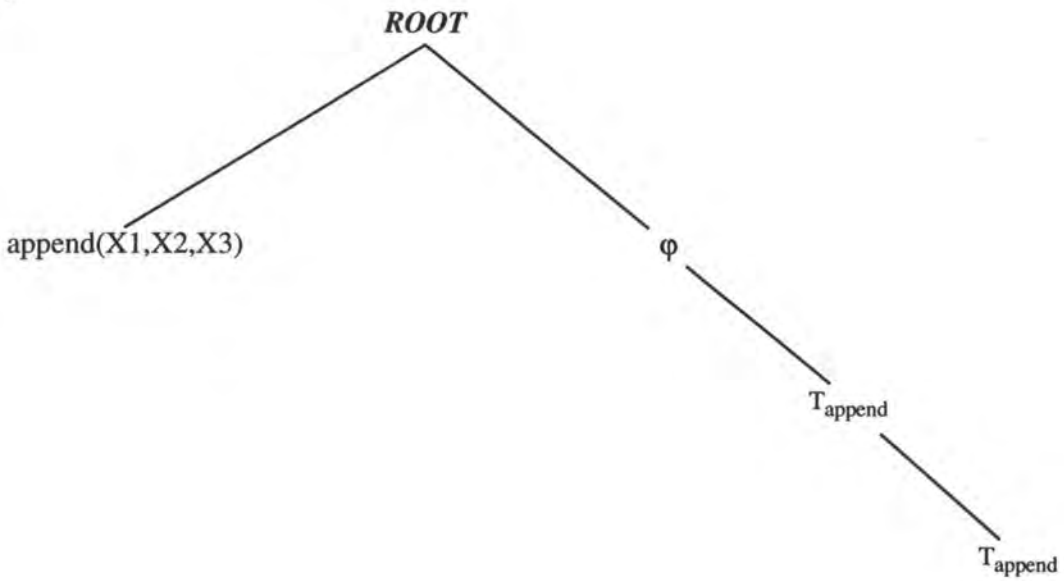


T_2 :



[5]Notons que cette suite sera toujours finie

T_3 :



A chaque itération l'évaluation de l'arbre T_{append} dernièrement ajouté permettra d'obtenir une meilleure approximation de la fonction eta du prédicat append.

C.2. Règles sémantiques.

Pour chaque production nous donnerons le graphe de dépendances et les règles sémantiques qui lui sont associées. Certaines productions ne sont présentes que pour des raisons purement syntaxiques amenées par la syntaxe de Prolog. C'est le cas des productions 4,6 et 7. Le but des règles sémantiques associées se limite donc à la propagation des attributs lors du parcours d'un arbre syntaxique. Pour cette raison ces productions ne nous inspirerons pas davantage de commentaires. Comme il a été dit, la fonction des règles sémantiques est double. D'une part, il s'agit de calculer les substitutions abstraites nécessaires à l'obtention des informations qui nous intéressent. D'autre part, la gestion des tables eta et anc associées à chaque prédicat doit servir de support tant au calcul de point fixe (suite croissante de fonctions eta) qu'au processus de memo-ization (terminaison du processus de calcul). Pour les productions concernées par cette double fonction nous ferons une série de commentaires afin de montrer comment chacune d'entre elles joue son rôle au sein de l'ensemble de la grammaire. En particulier nous tenterons d'établir un lien entre chaque règle sémantique et les idées exposées au second chapitre concernant tant l'interprétation abstraite proprement dite que la méthode de calcul récursif par memo-ization. Avant de présenter l'ensemble des règles sémantiques pour chaque production voici une brève description des attributs de la grammaire.

L'ensemble des attributs est le suivant :

$$A^*=\{\beta_{in},eta_{in},ich_{in},\beta_{out},eta_{out},ich_{out},anc,pred,lvar\}$$

- Les attributs β_{in} et β_{out} sont des substitutions abstraites à l'entrée et à la sortie du sous-arbre dont le noeud auquel ils sont attachés est la racine.

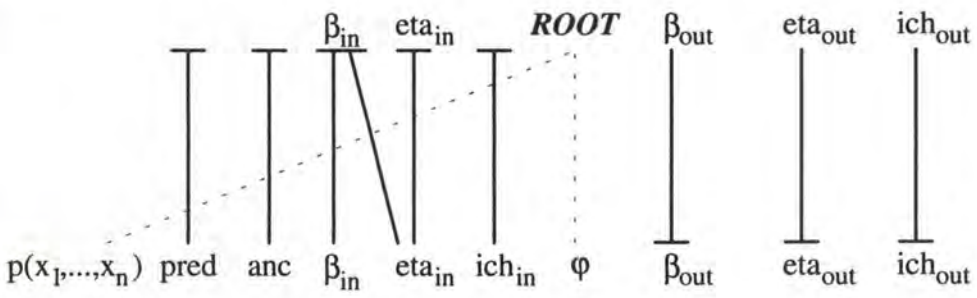
- eta_{in} et eta_{out} sont des tables de fonctions eta contenant des triplets (β_{in},p,β_{out}) pour lesquels $eta(\beta_{in},p)=\beta_{out}$.

- ich_{in} et ich_{out} sont des booléens qui sont positionnés à vrai tant que la table eta du noeud qui leur correspond n'a pas été modifiée. Leur utilité est de faciliter la détection du plus petit point fixe.

- anc est une table contenant des couples (β_{in},p). Lors de chaque passe du calcul de point fixe cette table contient l'ensemble des ocurrences de (β_{in},p) pour lesquelles une substitution abstraite en sortie à été calculée. Après chaque itération du calcul de point fixe cette table est remise à vide. Le but est de servir de support à la memo-ization. L'utilité de cette table deviendra évidente lorsque nous parlerons de l'algorithme de calcul de point fixe.

Voici maintenant les règles sémantiques associées à chacunes des onze productions de notre grammaire :

P1 :



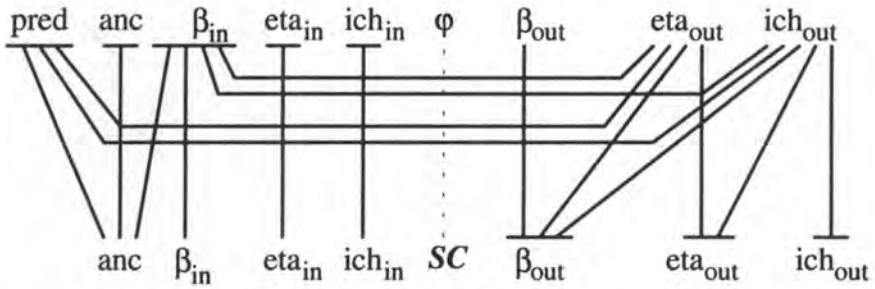
$\beta_{out}(ROOT)=\beta_{out}(\varphi)$
$eta_{out}(ROOT)=eta_{out}(\varphi)$
$ich_{out}(ROOT)=ich_{out}(\varphi)$

$anc(\varphi)=\varnothing$
$\beta_{in}(\varphi)=\beta_{in}(ROOT)$
$eta_{in}(\varphi)=Ext_p(eta_{in}(ROOT),(\beta_{in}(ROOT),p))$
$ich_{in}(\varphi)=true$
$pred(\varphi)=p$

Au second chapitre nous avions donné un exemple d'interprétation abstraite pour la fonction 91. Le processus de calcul de point fixe avait également été illustré au cours de ce même exemple. Chaque fois que le calcul de f([s..t]) pour s et t donnés était nécessaire on faisait dans un premier temps une approximation par défaut en considérant

qu'il s'agissait de la plus grande borne inférieure du domaine abstrait, en l'occurrence l'intervalle vide []. Le rôle de l'opérateur Ext_p ci-dessus est identique. On attribue par défaut la valeur $\perp$ à $eta(\beta_{in}(ROOT),p)$. Le triplet correspondant est stocké dans la table eta s'il y a lieu. Cette approximation par défaut pouvant être ensuite raffinée lors de l'itération suivante du calcul de point fixe.

P2 :



$\beta_{out}(\varphi)=\beta_{out}(SC)$

$eta_{out}(\varphi)=eta_{out}(SC)$ si $\beta_{out}(SC)\leq eta_{out}(SC)((\beta_{in}(\varphi),pred(\varphi)))$

$=Adj((\beta_{in}(\varphi),pred(\varphi),\beta_{out}(SC)),eta_{out}(SC))$ sinon

$ich_{out}(\varphi)=ich_{out}(SC)$ si $\beta_{out}(SC)\leq eta_{out}(SC)((\beta_{in}(\varphi),pred(\varphi)))$

$=false$ sinon

$anc(SC)=anc(\varphi) \cup (\beta_{in}(\varphi),pred(\varphi))$

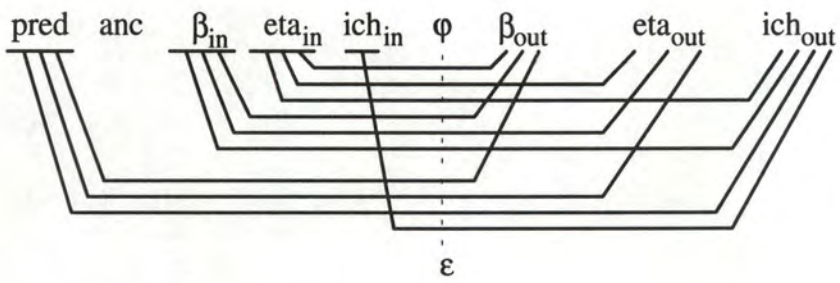
$\beta_{in}(SC)=\beta_{in}(\varphi)$

$eta_{in}(SC)=eta_{in}(\varphi)$

$ich_{in}(SC)=ich_{in}(\varphi)$

Dans la même ordre d'idée que ce que nous venons de dire concernant la production 1, la mise à jour de la table eta du symbole φ (opérateur Adj) vise à inclure dans la table une éventuelle amélioration de la substitution de sortie pour le couple $(\beta_{in}(\varphi),pred(\varphi))$ suite à l'évaluation du sous-arbre, dont φ est la racine, correspondant au prédicat pred(φ). Il s'agit bien d'une mise à jour éventuelle, c'est la raison pour laquelle on compare (opérateur ≤) $\beta_{out}(SC)$ et $eta_{out}(SC)((\beta_{in}(\varphi),pred(\varphi)))$. Le but principal étant de maintenir la cohérence de l'approximation que constitue le résultat stocké dans eta par rapport au résultat réel (donc sur le domaine concret) correspondant. On remarquera également que ayant calculé $eta(\beta_{in}(\varphi),pred(\varphi))$ on inclut le couple $(\beta_{in}(\varphi),pred(\varphi))$ dans la table eta pour éviter un recalcul de celui-ci par la suite. Il s'agit ici d'assurer le processus de memo-ization.

P3 :



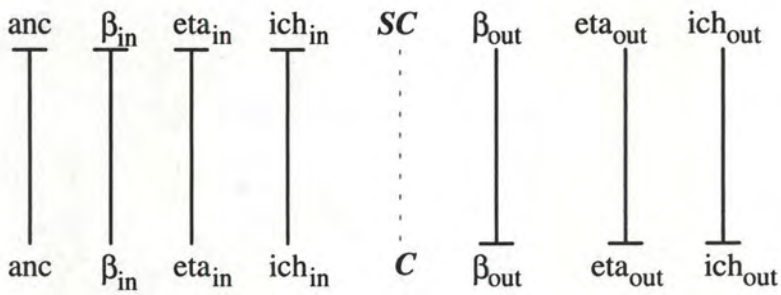
$\beta_{out}(\varphi)=eta_{in}(\varphi)(\beta_{in}(\varphi),pred(\varphi))$

$eta_{out}(\varphi)=eta_{in}(\varphi)$

$ich_{out}(\varphi)=ich_{in}(\varphi)$

L'unique effet de cette production est de consulter la table eta pour obtenir le résultat de $eta(\beta_{in}(\varphi),pred(\varphi))$. Celui-ci est ensuite propagé vers le haut dans l'arbre syntaxique.

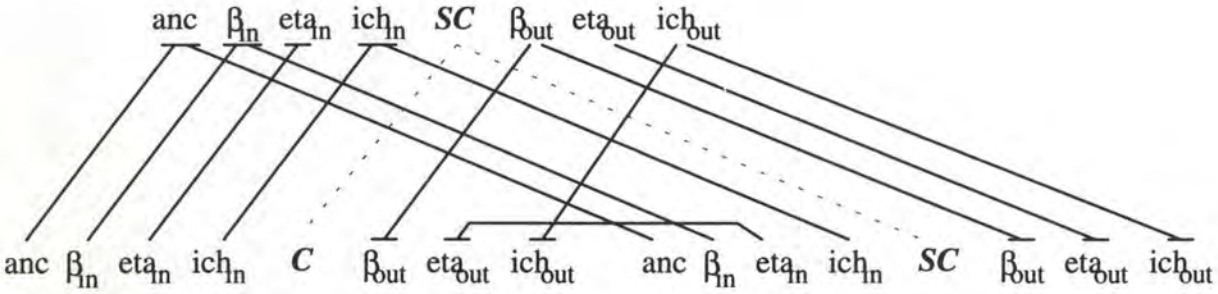
P4 :



$\beta_{out}(SC)=\beta_{out}(C)$
$eta_{out}(SC)=eta_{out}(C)$
$ich_{out}(SC)=ich_{out}(C)$

$anc(C)=anc(SC)$
$\beta_{in}(C)=\beta_{in}(SC)$
$eta_{in}(C)=eta_{in}(SC)$
$ich_{in}(C)=ich_{in}(SC)$

P5 :

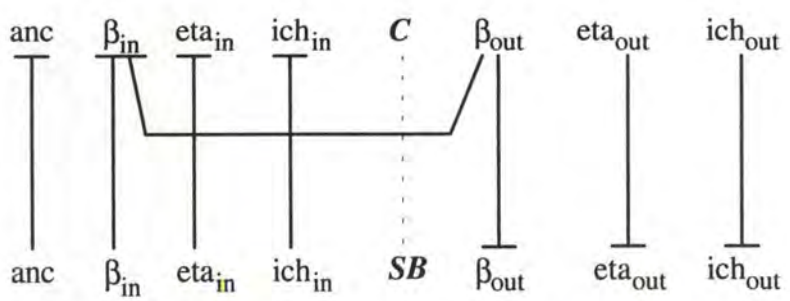


$\beta_{out}(SC_1)=\beta_{out}(C) \cup \beta_{out}(SC_2)$
$eta_{out}(SC_1)=eta_{out}(SC_2)$
$ich_{out}(SC_1)=ich_{out}(SC_2)$ and $ich_{out}(C)$

$anc(C)=anc(SC_1)$
$\beta_{in}(C)=\beta_{in}(SC_1)$
$eta_{in}(C)=eta_{in}(SC_1)$
$ich_{in}(C)=ich_{in}(SC_1)$
$anc(SC_2)=anc(SC_1)$
$\beta_{in}(SC_2)=\beta_{in}(SC_1)$
$eta_{in}(SC_2)=eta_{out}(C)$
$ich_{in}(SC_2)=ich_{in}(SC_1)$

Il faut ici remarquer le rôle de l'opérateur "$\cup$" dans la première équation sémantique. Dans l'exemple de la fonction 91 le lub de deux intervalles avait été défini comme le plus petit intervalle les recouvrant entièrement. L'opérateur "$\cup$" à la même fonction mais sur le domaine abstrait AS_D. Il s'agit de calculer le lub de deux substitutions abstraites. Nous avions déjà défini cet opérateur précédemment. La raison d'être de cette opération dans la cas présent est de "généraliser" en une seule substitution abstraite les résultats (deux substitutions abstraites en l'occurrence) fournis par l'interprétation abstraite de la clause représentée par le sous-arbre dont le noeud C est la racine et celle correspondant aux clauses du sous-arbre SC.

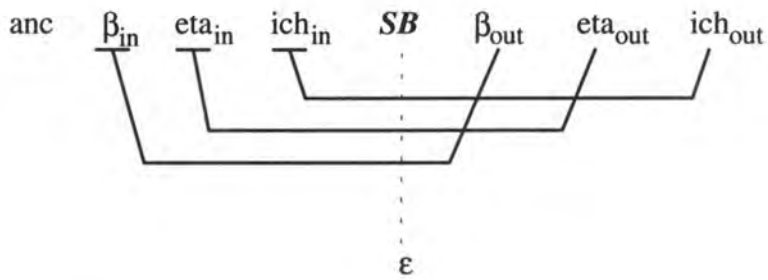
P6 :



$\beta_{out}(C)=Restr_c(\beta_{out}(SB),\beta_{in}(C))$
$eta_{out}(C)=eta_{out}(SB)$
$ich_{out}(C)=ich_{out}(SB)$

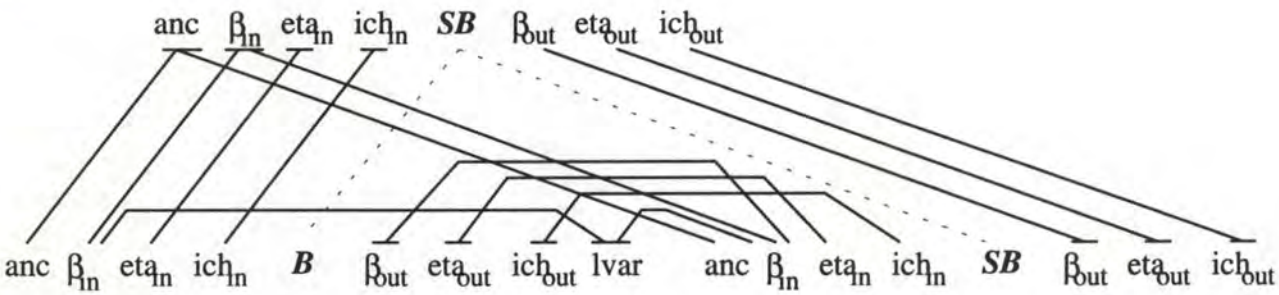
$anc(SB)=anc(C)$
$\beta_{in}(SB)=\beta_{in}(C)$
$eta_{in}(SB)=eta_{in}(C)$
$ich_{in}(SB)=ich_{in}(C)$

P7 :



$\beta_{out}(SB)=\beta_{in}(SB)$
$eta_{out}(SB)=eta_{in}(SB)$
$ich_{out}(SB)=ich_{in}(SB)$

P8 :

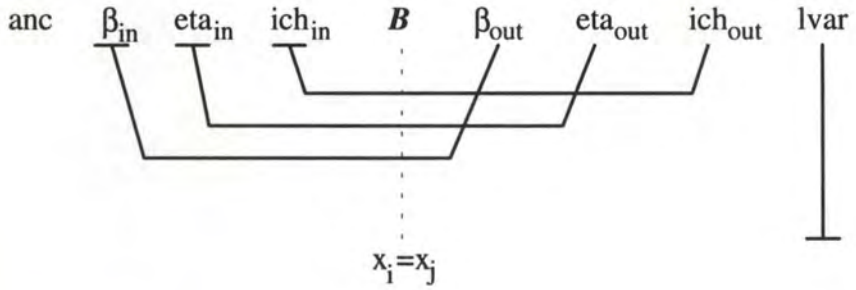


$\beta_{out}(SB_1)=\beta_{out}(SB_2)$
$eta_{out}(SB_1)=eta_{out}(SB_2)$
$ich_{out}(SB_1)=ich_{out}(SB_2)$

$anc(B)=anc(SB_1)$
$\beta_{in}(B)=Restr_b(\beta_{in}(SB_1),lvar(B))$
$eta_{in}(B)=eta_{in}(SB_1)$
$ich_{in}(B)=ich_{in}(SB_1)$
$anc(SB_2)=anc(SB_1)$
$\beta_{in}(SB_2)=Ext_b(lvar(B),\beta_{in}(SB_1),\beta_{out}(B))$
$eta_{in}(SB_2)=eta_{out}(B)$
$ich_{in}(SB_2)=ich_{out}(B)$

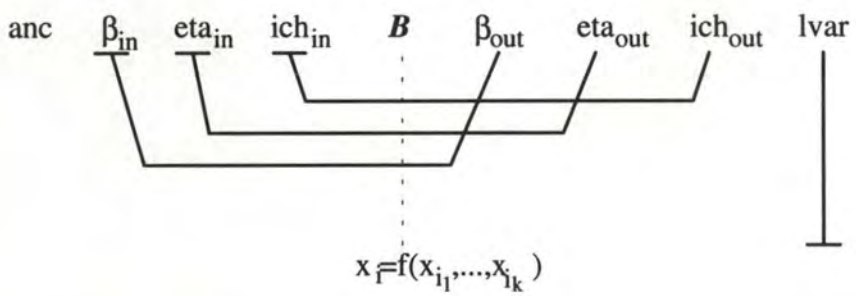
L'opérateur Ext_b est ici utilisé pour "inclure" à la substitution abstraite courante (celle attachée à SB_1 en entrée) l'autre substitution abstraite fournie par l'évaluation du sous-arbre B. Nous avons déjà précisé comment cet opérateur travaille.

P9 :



$\beta_{out}(B)=Abi_1(\beta_{in}(B))$
$eta_{out}(B)=eta_{in}(B)$
$ich_{out}(B)=ich_{in}(B)$
$lvar(B)=\{x_i,x_j\}$

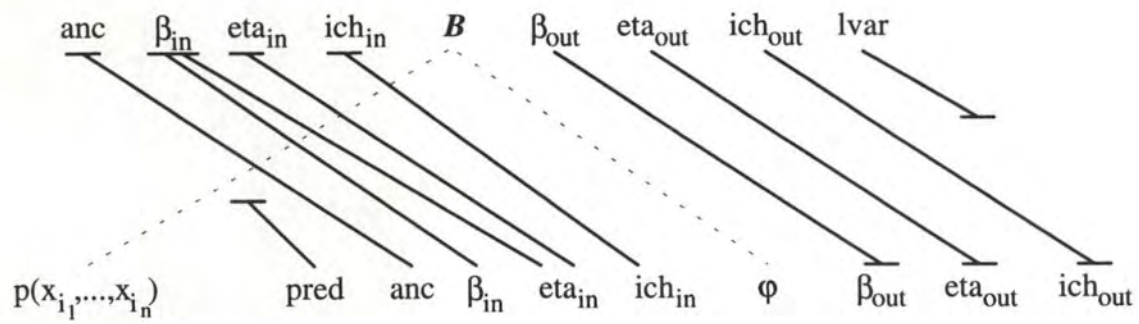
P10 :



$\beta_{out}(B)=Abi_2(\beta_{in}(B))$
$eta_{out}(B)=eta_{in}(B)$
$ich_{out}(B)=ich_{in}(B)$
$lvar(B)=\{x_i,x_{i_1},...,x_{i_k}\}$

Pour les productions 9 et 10, on remarquera l'utilisation des opérateurs Abi_1 et Abi_2 destinés à effectuer "abstraitement" les opérations associées aux buts représentés par les symboles terminaux de ces deux productions.

P11 :



$\beta_{out}(B)=Map((x_{i_1},...,x_{i_n}),\beta_{out}(\varphi),(x_1,...,x_n))$ $anc(\varphi)=anc(B)$
$eta_{out}(B)=eta_{out}(\varphi)$ $\beta_{in}(\varphi)=Map((x_1,...,x_n),\beta_{in}(B),(x_{i_1},...,x_{i_n}))$
$ich_{out}(B)=ich_{out}(\varphi)$ $eta_{in}(\varphi)=Ext_p(eta_{in}(B),(\beta_{in}(\varphi),p))$
$lvar(B)=(x_{i_1},...,x_{i_n})$ $ich_{in}(\varphi)=ich_{in}(B)$
$pred(\varphi)=p$

La production 11 est peut-être la plus intéressante de notre grammaire. En effet c'est elle qui symbolise les appels de prédicats pour les buts d'une clause. Il s'agit d'une part d'effectuer une sorte de changement de variables (au moyen de l'opérateur Map). La raison d'être de celui-ci est d'établir une relation entre les variables paramètres formels et les variables paramètres effectifs du prédicat appelé, et celà tant à l'entrée qu'à la sortie du noeud φ. D'autre part on retrouve comme précédemment (production 1) l'opérateur Ext_p destiné à la gestion de la table eta du prédicat appelé.

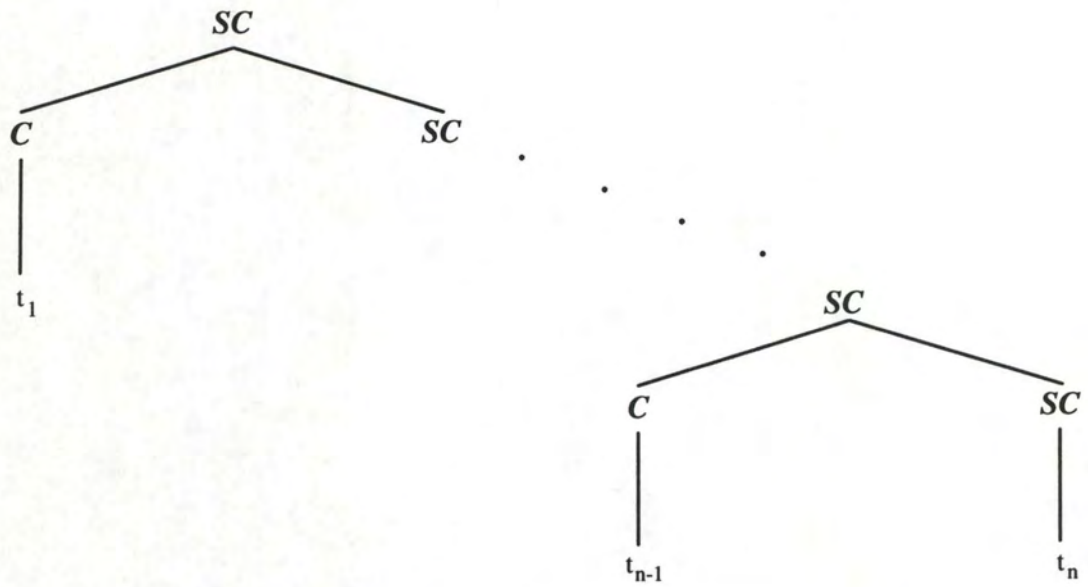
L'essentiel de ce qu'il faut retenir de la grammaire présentée est qu'elle définit la sémantique d'une transformation de fonctions qui va constituer la base du processus de calcul de point fixe. C'est ce dont nous allons parler maintenant.

D. Algorithme de calcul de point fixe.

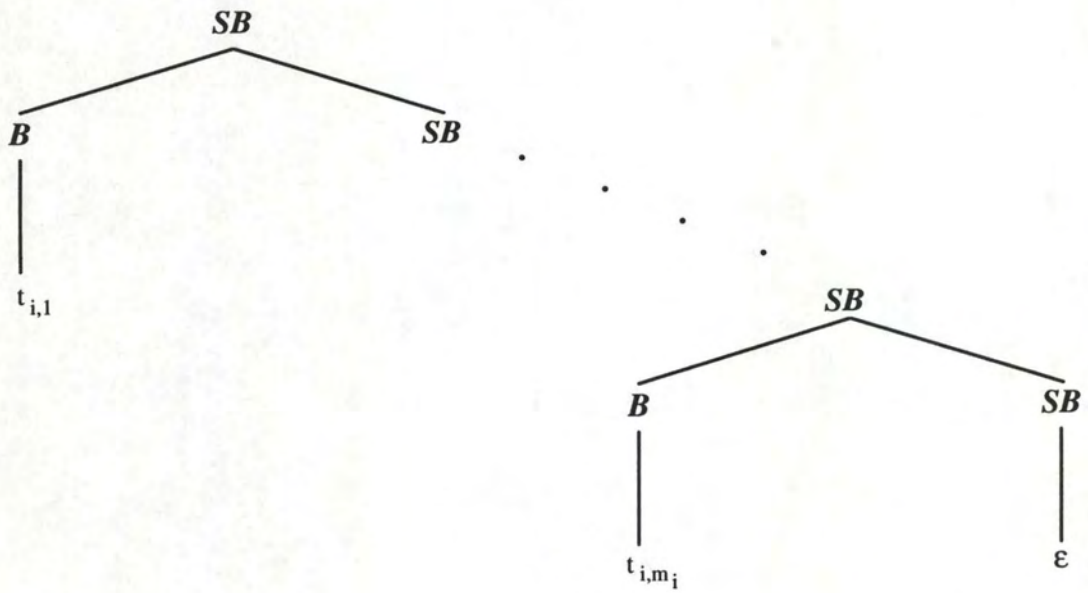
Lorsque nous avions parlé du plus petit point fixe d'une transformation de fonctions monotone continue, à la fin du deuxième chapitre, nous avions défini celui-ci comme étant la limite d'une suite croissante de fonctions engendrée par cette même transformation. Nous allons voir que la même idée se trouve à la base de l'algorithme de point fixe utilisé conjointement à la grammaire définie dans les deux sections précédentes. La fonction associée à un moment donné au programme Prolog étudié est donnée par la grammaire et plus précisément par la table eta attribut de la racine de l'arbre syntaxique construit pour ce programme. Le processus de calcul de point fixe consistera dès lors à construire une suite finie d'arbres syntaxiques (plus précisément un à chaque itération). Pour chaque arbre, la grammaire propagera la substitution initiale β_{in} ainsi que la table eta associée à la fonction équivalente au programme. Il s'agit là du processus de garnissage de l'arbre syntaxique.

La construction de la suite d'arbre syntaxique se déroulera en quatres étapes.

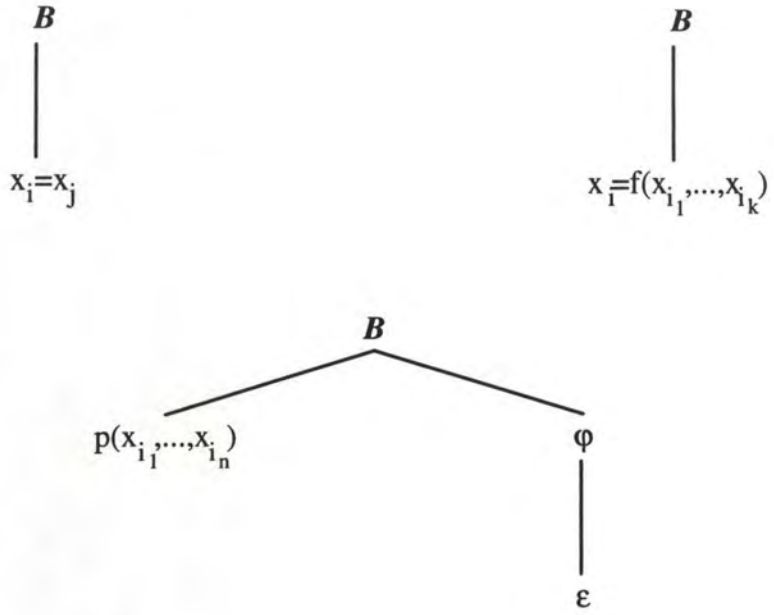
La première consistera à construire l'arbre syntaxique (sans calculer les attributs) de chaque prédicat défini dans le programme que l'on désire étudier. Ces arbres seront créés une fois pour toutes et réutilisés tels quels tant que le programme n'est pas modifié ou remplacé par un autre. La forme générale des arbres engendrés est la suivante :



Avec t_i $\forall$ i un arbre syntaxique de la forme :



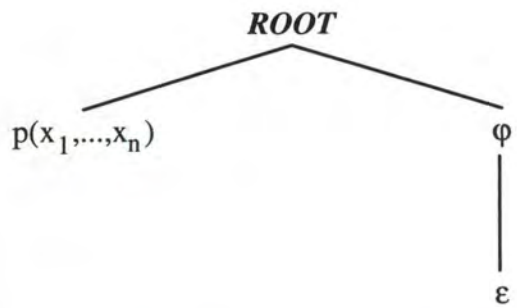
Et $\forall j : 1 \leq j \leq m_i$, $t_{i,j}$:



La seconde étape du calcul de point fixe consiste à développer l'arbre syntaxique pour la prochaine itération de celui-ci. Cette tâche est effectuée au moyen de l'algorithme suivant.

```
procédure Développer(t : arbre syntaxique)
  début
    pour chaque noeud u de t tel que u est de type φ et le fils de u est ε faire
      remplacer dans t le sous-arbre ε dont u est le père par T_p l'arbre
      syntaxique du prédicat p
  fin
```

Pour la première itération on initialisera l'arbre t comme suit :



A l'issue de cette première itération le résultat sera $\perp$ (production 3) puisque la table eta aura préalablement été initialisée à $eta(\beta_{in},p)=\perp$.

La transformation de l'arbre syntaxique effectuée par l'algorithme ci-dessus permettra lors de la prochaine évaluation des attributs de l'arbre d'améliorer l'approximation par défaut $\perp$ pour le couple $(\beta_{in}(\varphi),p)$.

L'étape suivante consiste à effectuer le calcul des attributs de l'arbre, son garnissage. A l'issue de l'évaluation la table eta du noeud racine de l'arbre définit une nouvelle fonction eta pour le prédicat associé. Cette nouvelle fonction constitue la fonction suivante de la suite croissante de fonctions pour le calcul de point fixe.

Les trois dernières étapes ci-dessus sont ensuite répétées jusqu'à l'obtention d'un plus petit point fixe. Etant donné ce que nous venons de dire concernant la suite des fonctions eta, celui-ci est atteint lorsque, à l'issue du processus d'évaluation des règles sémantiques de la grammaire, la table eta à la racine de l'arbre est restée inchangée. On peut dire dès lors que plus aucune amélioration de l'approximation fournie par β_{out}(ROOT) ne peut être espèrée.

Il ne nous reste plus maintenant qu'à donner la forme explicite de l'algorithme de calcul de point fixe. L'algorithme calcule le plus petit point fixe pour un prédicat p est une substitution abstraite β donnés.

```
procédure calc_pt_fixe(p,β)
  début
    initialiser t;
    β_in(ROOT)←β;
    eta_in(ROOT)←∅;
    itération(t);
  fin

procédure itération(t : arbre)
  début
    eval(t);   /* évaluation des attributs de l'arbre syntaxique */
    si ich_out(ROOT)=true    /* c'est-à-dire si
                              eta_in(ROOT)=eta_out(ROOT) */
    alors
      début
        eta_out(ROOT) est le plus petit point fixe et eta(β_in(ROOT),p)=
        β_out(ROOT);
        stop;
      fin;
    sinon
      début
        eta_in(ROOT)←eta_out(ROOT);
        développer(t);
        itération(t);
      fin
  fin
```

```
procédure développer(t : arbre)
  début
    pour chaque noeud u de t tel que u est de type φ et le fils de u est ε et
    (β_in(u),pred(u))∉ anc(u) faire
      remplacer dans t le sous-arbre ε dont u est le père par T_p l'arbre
      syntaxique du prédicat p;
  fin
```

Nous ferons une dernière remarque concernant l'utilisation de la table anc dans la procédure développer. Nous avions déjà dit que lors de chaque passe du calcul de point fixe cette table contient l'ensemble des occurrences de (β_{in},p) pour lesquelles une substitution abstraite en sortie à été calculée. Son utilisation permet donc de ne pas effectuer plusieurs fois le même calcul, ce qui est important pour le calcul récursif descendant, ainsi que d'assurer que l'arbre syntaxique ne se développe pas indéfiniment ce qui serait une cause de bouclage pour notre algorithme.

E. Un exemple simple.

Afin d'illustrer la manière dont fonctionne l'interpréteur abstrait que nous venons de présenter, nous allons donner un exemple simple d'application de celui-ci. Pour ce faire, nous allons à nouveau considérer le programme append. Le but de ce programme est d'effectuer la concaténation de deux listes. Celui-ci est particulièrement adapté puisqu'il va nous permettre de donner une application de la méthode qui soit courte et simple ce qui en accroît la lisibilité. Rappelons comment était défini le prédicat append.

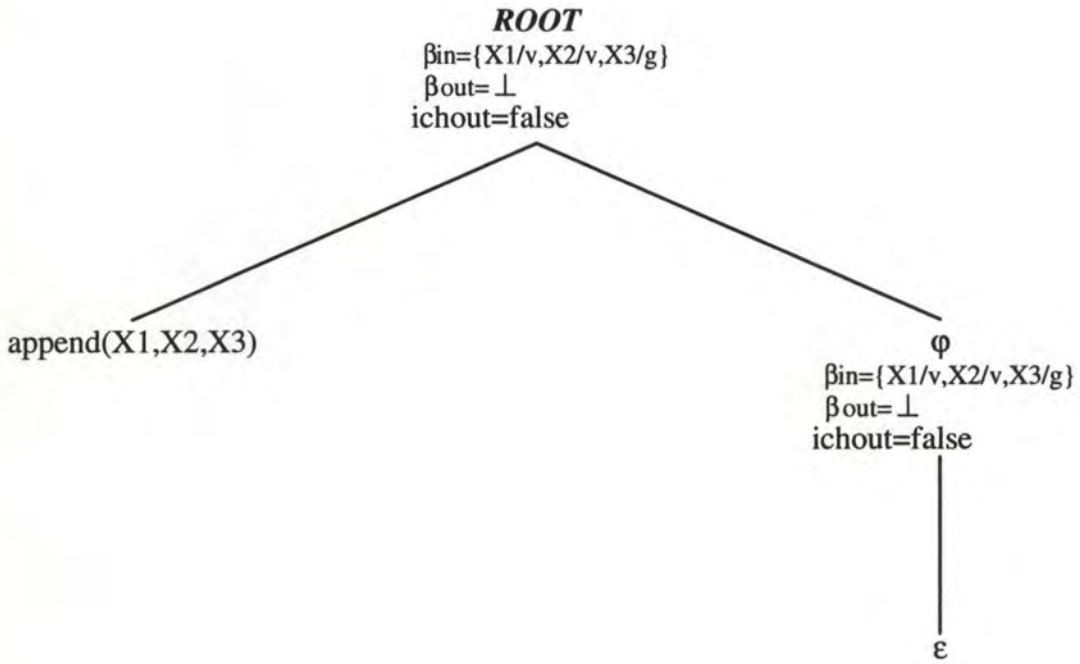
```
append(X1,X2,X3):-X1=[],X2=X3.
append(X1,X2,X3):-X3=[X4|X6],append(X5,X2,X6),X1=[X4|X5].
```

L'exemple consiste à donner le détail du calcul du plus petit point fixe pour le prédicat append avec {X1/var,X2/var,X3/ground} comme substitution abstraite en entrée. Le plus petit point fixe est atteint après trois itérations. La substitution abstraite en sortie est alors {X1/ground,X2/ground,X3/ground}. Pour chacune des deux premières itérations nous donnons l'arbre syntaxique utilisé annoté des attributs β_{in}, β_{out} et ich_{out}. Nous donnons également les tables eta et anc respectivement à la racine et au noeud φ feuille pour chaque itération. On remarque le changement de la table eta entre les deux premières itérations : l'approximation de eta({X1/var,X2/var,X3/ground},append) est améliorée. A la fin de la première itération, la table anc du noeud φ correspondant à l'appel récursif du prédicat append est vide. L'arbre syntaxique est donc étendu. A la fin de la deuxième itération, on constate que cette même table contient le couple ({X1/var,X2/var,X3/ground},append). On a que $(\beta_{in}(\varphi),pred(\varphi))\in anc(\varphi)$[6]. L'arbre syntaxique n'est de ce fait pas étendu. C'est la raison pour laquelle nous n'avons pas

[6]Rappelons que la table anc contient l'ensemble des couples (β_{in},p) déjà calculés préalablement

donné le détail de la troisième itération. L'arbre syntaxique n'est pas modifié au cours de celle-ci, pas plus que la valeur des attributs[7]. Cette troisième itération est uniquement effectuée pour s'apercevoir que la table eta du prédicat append n'a pas été modifiée et que donc le plus petit point fixe est atteint.

ITERATION 1:

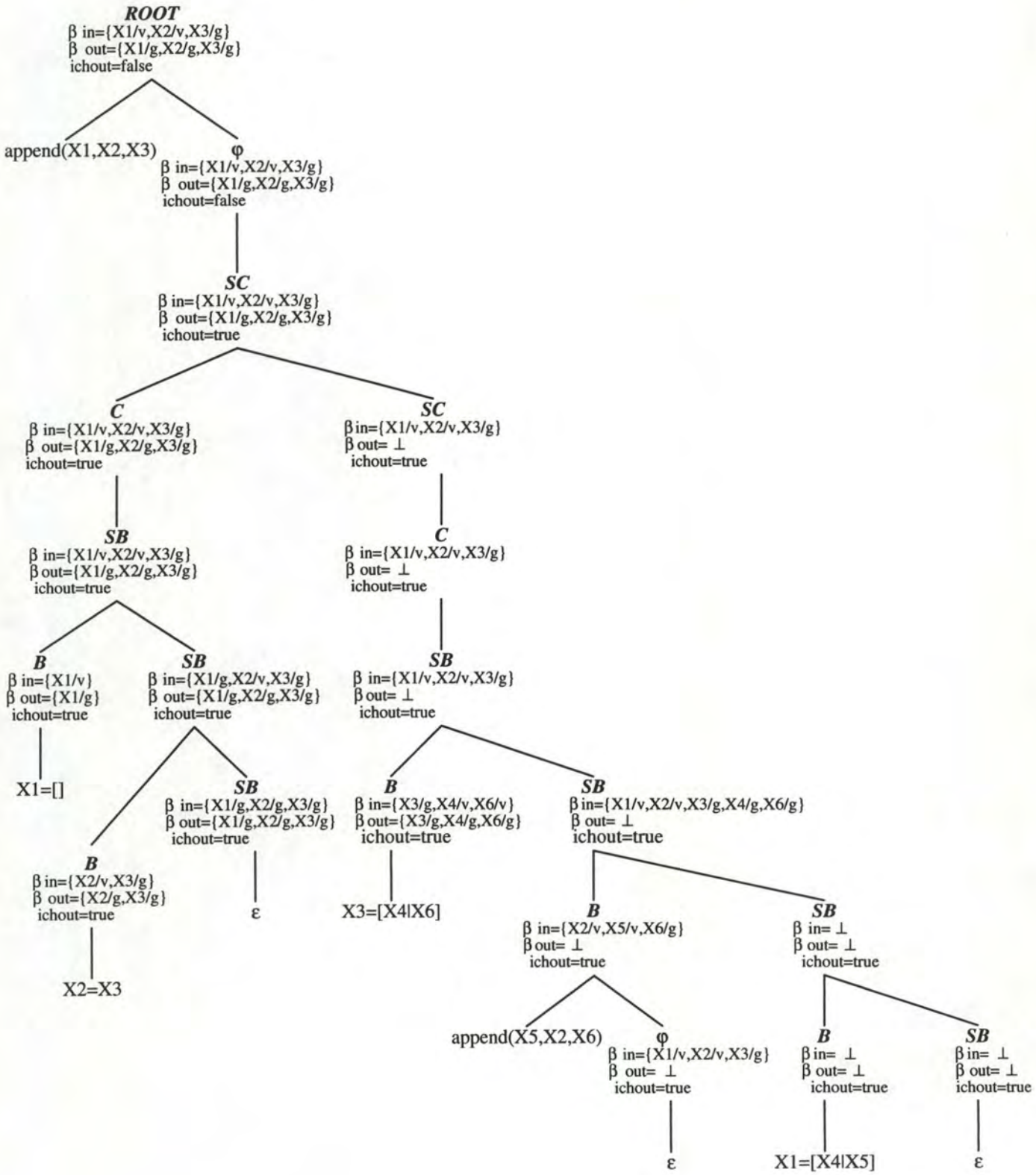


Tables :

eta_{out}(ROOT)={({X1/var,X2/var,X3/ground},append,⊥)}
anc(φ)={}

[7]A l'exception des attributs ich_{in} et ich_{out}

ITERATION 2:


ROOT
β in={X1/v,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=false
append(X1,X2,X3)
φ
β in={X1/v,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=false
SC
β in={X1/v,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=true
C
β in={X1/v,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=true
SC
β in={X1/v,X2/v,X3/g}
β out= ⊥
ichout=true
SB
β in={X1/v,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=true
C
β in={X1/v,X2/v,X3/g}
β out= ⊥
ichout=true
B
β in={X1/v}
β out={X1/g}
ichout=true
SB
β in={X1/g,X2/v,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=true
SB
β in={X1/v,X2/v,X3/g}
β out= ⊥
ichout=true
X1=[]
B
β in={X1/g,X2/g,X3/g}
β out={X1/g,X2/g,X3/g}
ichout=true
B
β in={X3/g,X4/v,X6/v}
β out={X3/g,X4/g,X6/g}
ichout=true
SB
β in={X1/v,X2/v,X3/g,X4/g,X6/g}
β out= ⊥
ichout=true
B
β in={X2/v,X3/g}
β out={X2/g,X3/g}
ichout=true
ε
X3=[X4|X6]
B
β in={X2/v,X5/v,X6/g}
β out= ⊥
ichout=true
SB
β in= ⊥
β out= ⊥
ichout=true
X2=X3
append(X5,X2,X6)
φ
β in={X1/v,X2/v,X3/g}
β out= ⊥
ichout=true
B
β in= ⊥
β out= ⊥
ichout=true
SB
β in= ⊥
β out= ⊥
ichout=true
ε
X1=[X4|X5]
ε

Tables :

eta_{out}(ROOT)={({X1/var,X2/var,X3/ground},append,{X1/ground,X2/ground, X3/ground})}

anc(φ)={({X1/var,X2/var,X3/ground},append)}

CHAPITRE V : L'INTERPRETEUR ABSTRAIT AAIAG

Chapitre V : l'interpréteur abstrait AAIAG[1].

A. L'interpréteur abstrait.

A.1. Utilisation et restrictions applicables.

Le programme présenté dans cette section n'est autre que l'implémentation de l'interpréteur abstrait décrit au quatrième chapitre. Nous avons réalisé cette implémentation au cours d'un stage effectué au LaBRI[2] d'octobre à décembre 93. Nous allons commencer par en expliquer le mode d'utilisation. Son application au programme "append" va nous servir d'exemple. Le programme Prolog en question était :

```
append(X1,X2,X3):-X1=[],X2=X3.
append(X1,X2,X3):-X3=[X4|X6],append(X5,X2,X6),X1=[X4|X5].
```

Remarquons que la syntaxe du programme a été mise sous une forme admissible par l'interpréteur. La normalisation d'un programme Prolog en vue de le rendre interprétable par AAIAG peut être effectuée automatiquement au moyen du programme NP dont il sera question dans une des sections suivantes. Pour appliquer l'interpréteur abstrait à "append", exécuter la commande suivante :

```
aaiag append.pro /v
```

"append.pro" est le nom du fichier contenant le texte du programme. Le paramètre "/v" est optionnel. Il commande à l'interpréteur de fournir la trace complète des attributs des arbres syntaxiques utilisés pour chacune des itérations du calcul de point fixe.

Une fois l'exécution commencée, le programme demande d'abord à l'utilisateur d'introduire le nom du prédicat qui doit être interprété. Dans notre exemple, il s'agira de "append". Le programme demande ensuite de fournir une substitution abstraite d'entrée (β_{in}) pour le prédicat considéré. Nous donnons ci-dessous le dialogue complet de l'interpréteur et de l'utilisateur pour l'exemple du programme append. Les termes en italique doivent être introduits par l'utilisateur. Le programme fournit comme résultats les substitutions abstraites en entrée et en sortie, le nombre d'itérations pour le calcul de point fixe et le nombre total de sommets évalués pour l'ensemble des arbres syntaxiques engendrés.

```
>aaiag append.pro
Prédicat d'appel : append
X1/var
```

[1] An Abstract Interpreter by Attributed Grammar
[2] Laboratoire Bordelais de Recherche en Informatique

X2/*var*
X3/*ground*

Bin={X1/var,X2/var,X3/ground}
Bout={X1/ground,X2/ground,X3/ground}
3 itérations effectuées
40 sommets parcourus

La syntaxe admise pour le texte du programme que l'on désire étudier, est semblable à celle définie par la grammaire du chapitre précédent. Les différences entre les deux sont mineures. Par exemple la grammaire du chapitre 4 définit un type de but "$x_i=f(x_{i_1},...,x_{i_k})$". La syntaxe exacte admise par l'interpréteur est la suivante. Le symbole f peut être soit un foncteur quelconque d'arité n (n>0) avec des variables comme paramètres, soit une constante numérique ou symbolique c'est-à-dire composée de lettres et de chiffres avec une lettre en tête. Par souci de précision nous donnons ici une grammaire définissant la syntaxe admise par AAIAG.

SC*→*C* | *C SC
***C*→*TC*:-*SB*. | *TC*.**
SB*→*B* | *B*,*SB
B*→**VAR=VAR | VAR=FC* | *TC***
TC*→**NPF(LVAR***)**
FC*→**NPF | CST | NPF(LVAR***) | [VAR|VAR] | []**
LVAR*→**VAR | VAR,LVAR***

Les symboles terminaux VAR, CST et NPF sont définis par les expressions régulières suivantes.

VAR : [A..Z] ([a..z] | [A..Z] | [0..9] | _)*
CST : [0..9]+
NPF : [a..z] ([a..z] | [A..Z] | [0..9] | _)*

Nous donnons également en annexe le schéma de traduction Lex et Yacc utilisé pour l'implémentation de la grammaire ci-dessus.

Signalons encore une restriction d'ordre pratique concernant la syntaxe des clauses de tout programme admissible. Les têtes de clauses pour un même prédicat doivent être déclarées en utilisant les mêmes variables. Par exemple le programme suivant sera refusé par l'interpréteur.

q(X1,X2,X3):-...
q(X1,X3,X4):-...

Alors que le programme suivant sera accepté.

```
q(X1,X2,X3):-...
q(X1,X2,X3):-...
```

La raison d'être d'une telle restriction, par ailleurs peu contraignante, est de simplifier la gestion des variables utilisées par un prédicat.

A.2. Structure et mode de fonctionnement.

L'implantation du programme AAIAG a été réalisée en langage C interfacé avec un schéma de traduction Lex et Yacc. Le programme est composé de 8 modules différents dont nous donnons la liste ainsi qu'un bref commentaire concernant leurs rôles respectifs. Le mode de calcul de point fixe ainsi que le processus d'évaluation des attributs de la grammaire sont identiques à ceux présentés au quatrième chapitre. Le parcours des sommets d'un arbre syntaxique est effectué en profondeur d'abord. L'évaluation des attributs de la grammaire a lieu en deux temps en raison de la présence d'attributs hérités et synthétisés. Lors de la descente dans l'arbre les valeurs des attributs hérités sont propagées vers le bas. Lors de la remontée, les valeurs des attributs hérités peuvent alors être calculées.

gramabs.c	Implémentation des règles sémantiques de la grammaire et de l'ordre de parcours en profondeur d'abord pour l'évaluation des attributs d'un arbre syntaxique.
trees.c	Procédures de création et de gestion des arbres syntaxiques des prédicats du programme étudié ainsi que d'extension de l'arbre après chaque itération du calcul de point fixe.
trees.lex trees.yacc	Schémas de traduction pour l'implémentation de la grammaire définissant la syntaxe d'entrée pour le texte du programme Prolog. Utilisé conjointement à trees.c pour la création des arbres syntaxiques des prédicats du programme Prolog.
view.c	Module de visualisation des attributs d'un arbre syntaxique annoté pour la trace d'exécution.
aaiag.c	Module principal. Coordonne l'activité des autres modules.
opabs.c	Implémentation des opérateurs abstraits et du domaine des modes (module typabs.h).
utils.c	Procédures diverses concernant notamment la gestion des tables eta et anc ainsi que des listes de variables.

Le code des différents modules est donné en annexe.

B. Le préprocesseur NP.

Le but du préprocesseur NP est de fournir une forme admissible par l'interpréteur AAIAG pour un programme Prolog donné. Afin d'illustrer l'action de ce programme nous allons l'appliquer au programme append dans une de ses versions courantes.

```
append([],X,Y):-X=Y.
append([X4|X5],X2,[X4|X6]):-append(X5,X2,X6).
```

La commande d'appel correspondante est :

```
np append.src
```

NP produit deux fichiers en sortie : append.tmp et append.nor. Le fichier .tmp contient une version intermédiaire du programme normalisé. Nous en donnons le listing ci-dessous.

```
append(YY0,X,Y):-YY0=[],X=Y.
append(YY1,X2,YY2):-YY1=[X4|X5],YY2=[X4|X6],append(X5,X2,X6),
YY1=[X4|X5],YY2=[X4|X6].
```

On remarque plusieurs choses concernant le programme ci-dessus.

- Le préprocesseur a généré des nouvelles variables (YY) afin d'aplatir les buts et en-têtes de clauses non conformes à la syntaxe admissible par l'interpréteur.

- Les buts "YY1=[X4|X5]" et "YY2=[X4|X6]" ont été répétés après l'appel au prédicat append. Le but de cette répétition est de minimiser les imprécisions liées à l'interprétation abstraite avec le domaine des modes. Si l'on avait interprété le prédicat append avec la substitution abstraite {X1/var,X2/var,X3/ground} en entrée sans effectuer la répétition de buts, on aurait obtenu {X1/any,X2/ground,X3/ground} comme résultat. Cela est correct mais imprécis, la meilleure approximation possible étant {X1/ground,X2/ground,X3/ground}. C'est précisément le résultat fourni pour le programme incluant les répétitions de buts. De manière générale, le préprocesseur génère une répétition pour chaque but autre qu'un appel de prédicat et celà immédiatement après l'appel de prédicat suivant. De manière plus précise, toute clause de la forme :

  $\ldots, B_{i-2}, B_{i-1}, B_i, p(x_{i_1},\ldots,x_{i_n}), B_{i+1}, \ldots$

  devient :

$$\ldots, B_{i-2}, B_{i-1}, B_i, p(x_{i_1},\ldots,x_{i_n}), B_{i-2}, B_{i-1}, B_i, B_{i+1}, \ldots$$

avec B_{i-2}, B_{i-1}, B_i et B_{i+1} des buts de type "$x_i=x_j$" ou "$x_i=f(x_{i_1},\ldots,x_{i_k})$" et $p(x_{i_1},\ldots,x_{i_n})$ un appel de prédicat. Une telle manière de procéder ne modifie en rien la sémantique du programme interprété. Le lecteur intéressé trouvera dans [7] une étude de cette méthode de répétition de buts.

Il est également possible d'obtenir un programme normalisé dans lequel les répétitions de buts n'ont pas été effectuées. Il suffit pour cela d'invoquer le préprocesseur en plaçant après le nom du fichier source le paramètre "/n".

• Le programme ci-dessus n'est cependant pas encore admissible par l'interpréteur AAIAG. Nous avions précisé que les en-têtes de clauses concernant un même prédicat devaient utiliser les mêmes variables. Ce n'est pas le cas ici. Afin de remédier à ce problème NP effectue une phase finale de changement de variables. Dans le cas de notre exemple le fichier append.nor contient le programme ainsi transformé. C'est lui qui doit être fourni comme source à l'interpréteur. Voici le listing du fichier append.nor.

```
append(YY3,YY4,YY5):-YY3=[],YY4=YY5.
append(YY3,YY4,YY5):-YY3=[X4|X5],YY5=[X4|X6],append(X5,YY4,X6),
YY3=[X4|X5],YY5=[X4|X6].
```

De la même manière que pour l'interpréteur abstrait, nous donnons la grammaire définissant la syntaxe que doit respecter tout texte d'entrée pour le préprocesseur NP.

SC*→*C SC* | *C
***C*→*TCAP*:-*SB*.** | ***TCAP*.**
SB*→*B*,*SB* | *B
B*→*TCAP | $VAR_1=VAR_2$ | VAR=***FC***
FC*→**NPF(LARG*) | NPF | [VAR_1|VAR_2] | [] | CST
LARG*→*ARG*,*LARG* | *ARG
ARG*→**VAR | [VAR_1|VAR_2] | [] | NPF(LARG*) | NPF | CST

Les symboles terminaux VAR, NPF et CST ont la même syntaxe que celle définie précédemment pour l'interpréteur abstrait. Faisons remarquer que les foncteurs et prédicats[3] en notation infixée tels que "<" ou "+" seront refusés. Ceux-ci doivent donc être transformés manuellement en un équivalent en notation préfixée.

Signalons encore que l'implémentation de NP à été effectuée en langage C. Nous donnons en annexe le code correspondant ainsi que le schéma de traduction utilisé comme base pour l'implémentation.

[3]A l'exception de "="

C. Tests et évaluation.

Dans cette dernière section nous allons illustrer les résultats fournis par la méthode d'interprétation abstraite par grammaire attribuée avec le domaine des modes. Pour ce faire l'interpréteur abstrait a été appliqué à trois programmes différents.

Append : le programme déja utilisé à plusieurs reprises.
Quicksort : la méthode de tri bien connue.
Queens : le problème des huits reines.

Pour chacun des trois programmes nous donnerons le programme source, le programme normalisé et un tableau présentant les différents tests effectués. Les tableaux sont composés de dix colonnes dont voici la liste accompagnée d'une description du résultat fourni.

- β_{in} : substitution abstraite en entrée.
- T_r : temps de calcul avec les répétitions de buts (en secondes).
- I_r : nombre d'itérations du processus de calcul de point fixe avec les répétitions de buts.
- S_r : nombre de sommets parcourus dans les arbres syntaxiques avec les répétitions de buts.
- β_{outr} : substitution abstraite en sortie avec les répétitions de buts.
- T_n : temps de calcul sans les répétitions de buts (en secondes).
- I_n : nombre d'itérations du processus de calcul de point fixe sans les répétitions de buts.
- S_n : nombre de sommets parcourus dans les arbres syntaxiques sans les répétitions de buts.
- β_{outn} : substitution abstraite en sortie sans les répétitions de buts.
- DS : taux d'accroissement du nombre de sommets parcourus avec les répétitions de buts.

Pour chaque programme nous avons choisi les substitutions abstraites β_{in} qui correspondaient aux modes couramment utilisés.

C.1. Test Append.

Programme source :

```
append([],X,Y):-X=Y.
append([X4|X5],X2,[X4|X6]):-append(X5,X2,X6).
```

Programme normalisé :

```
append(YY3,YY4,YY5):-YY3=[],YY4=YY5.
append(YY3,YY4,YY5):-YY3=[X4|X5],YY5=[X4|X6],append(X5,YY4,X6),
YY3=[X4|X5],YY5=[X4|X6].
```

Tableau des résultats :

β_{in}	T_r	I_r	S_r	β_{outr}	T_n	I_n	S_n	β_{outn}	DS
{g,g,g}	0.1	3	48	{g,g,g}	0.1	3	40	{g,g,g}	0.2
{v,v,g}	0.1	3	48	{g,g,g}	0.1	4	59	{a,g,g}	-0.19
{g,g,v}	0.1	3	48	{g,g,g}	0.1	4	59	{g,g,a}	-0.19
{v,v,v}	0.1	4	71	{a,v,a}	0.1	4	59	{a,v,a}	0.2

On constate ici que la répétition statique de buts est une technique payante. En particulier pour les substitutions en entrée {v,v,g} et {g,g,v} (des modes par ailleurs très courants dans la pratique) celle-ci permet d'améliorer conjointement la performance du calcul de point fixe (-19% de sommets parcourus et une itération de moins) et la précision des résultats obtenus. Seul le cas de {v,v,v} s'avère pénalisant[4].

C.2. Test Quicksort.

Programme source :

```
quicksort(V1,V2):-V1=[],V2=[].
quicksort(V1,V2):-V1=[V3|V4],partition(V5,V3,V6,V7),quicksort(V6,V8),
                  quicksort(V7,V9), V10=[V3|V9],append(V8,V10,V2).

partition(V1,V2,V3,V4):-V1=[],V3=[],V4=[].
partition(V1,V2,V3,V4):-V1=[V5|V6],V3=[V5|V7],V5=V2,
                         partition(V6,V2,V7,V4).
partition(V1,V2,V3,V4):-V1=[V5|V6],V4=[V5|V7],V5=V2,
                         partition(V6,V2,V3,V7).

append(V1,V2,V3):-V1=[],V3=V2.
append(V1,V2,V3):-V1=[V4|V5],V3=[V4|V6],append(V5,V2,V6).
```

Programme normalisé :

```
append(YY0,YY1,YY2):-YY0=[],YY2=YY1.
```

[4]Notons que celui-ci n'est pratiquement jamais utilisé

```
append(YY0,YY1,YY2):-YY0=[V4|V5],YY2=[V4|V6],append(V5,YY1,V6),
                     YY0=[V4|V5],YY2=[V4|V6].

partition(YY3,YY4,YY5,YY6):-YY3=[],YY5=[],YY6=[].
partition(YY3,YY4,YY5,YY6):-YY3=[V5|V6],YY5=[V5|V7],V5=YY4,
                            partition(V6,YY4,V7,YY6),YY3=[V5|V6],
                            YY5=[V5|V7],V5=YY4.
partition(YY3,YY4,YY5,YY6):-YY3=[V5|V6],YY6=[V5|V7],V5=YY4,
                            partition(V6,YY4,YY5,V7),YY3=[V5|V6],
                            YY6=[V5|V7],V5=YY4.

quicksort(YY7,YY8):-YY7=[],YY8=[].
quicksort(YY7,YY8):-YY7=[V3|V4],partition(V5,V3,V6,V7),YY7=[V3|V4],
                    quicksort(V6,V8),quicksort(V7,V9),V10=[V3|V9],
                    append(V8,V10,YY8), V10=[V3|V9].
```

Tableau des résultats :

β_{in}	T_r	I_r	S_r	β_{outr}	T_n	I_n	S_n	β_{outn}	DS
{g,g}	0.6	5	652	{g,g}	1.3	8	1659	{g,g}	-0.6
{g,v}	0.4	4	350	{g,g}	1.2	7	1462	{g,a}	-0.76
{v,g}	2.2	7	2384	{a,g}	1.7	7	1948	{a,g}	0.22
{v,v}	1.9	7	2048	{a,a}	1.4	7	1676	{a,a}	0.22

Les résultats fournis par les répétitions de buts restent concluants dans le cas de quicksort. On constate que la méthode permet d'obtenir des gains de performances importants (-60% et -76%) pour le nombre de sommets évalués. La répétition de buts n'apporte rien dans le cas du mode {v,g}. Pours le mode {g,v}, qui est le plus couramment utilisé, on constate une nette amélioration.

C.3. Test Queens.

Programme source :

```
queens(X,Y) :-perm(X,Y),safe(Y).

perm([],[]).
perm([X|Y],[V|Res]) :-delete(V,[X|Y],Rest),perm(Rest,Res).

delete(X,[X|Y],Y).
delete(X,[F|T],[F|R]) :-delete(X,T,R).
```

```
safe([]).
safe([X|Y]) :-noattack(X,Y,1),safe(Y).

noattack(X,[],N).
noattack(X,[F|T],N) :-notequ(X,F),notequ(X,add(F,N)),notequ(F,add(X,N)),
                      noattack(X,T,N1), load(N1,add(N,1)).

notequ(X,Y):-X=a,Y=a.

load(X,Y):-X=a,Y=a.
```

Programme normalisé :

```
delete(YY17,YY18,YY19):-YY18=[YY17|YY19].
delete(YY17,YY18,YY19):-YY18=[F|T],YY19=[F|R],delete(YY17,T,R),
                        YY18=[F|T], YY19=[F|R].

load(YY20,YY21):-YY20=a,YY21=a.

noattack(YY22,YY23,YY24):-YY23=[].
noattack(YY22,YY23,YY24):-YY23=[F|T],notequ(YY22,F),YY23=[F|T],
                          YY13=add(F,YY24),notequ(YY22,YY13),
                          YY13=add(F,YY24),YY14=add(YY22,YY24),
                          notequ(F,YY14),YY14=add(YY22,YY24),
                          noattack(YY22,T,N1),YY16=1,
                          YY15=add(YY24,YY16),load(N1,YY15),
                          YY16=1,YY15=add(YY24,YY16).

notequ(YY25,YY26):-YY25=a,YY26=a.

perm(YY27,YY28):-YY27=[],YY28=[].
perm(YY27,YY28):-YY27=[X|Y],YY28=[V|Res],YY4=[X|Y],
                 delete(V,YY4,Rest),YY27=[X|Y], YY28=[V|Res],
                 YY4=[X|Y],perm(Rest,Res).

queens(YY29,YY30):-perm(YY29,YY30),safe(YY30).

safe(YY31):-YY31=[].
safe(YY31):-YY31=[X|Y],YY10=1,noattack(X,Y,YY10),YY31=[X|Y],YY10=1,
            safe(Y).
```

Tableau des résultats :

β_{in}	T_r	I_r	S_r	β_{outr}	T_n	I_n	S_n	β_{outn}	DS
{g,g}	0.9	7	1114	{g,g}	1.2	9	1612	{g,g}	-0.3
{g,v}	1.1	7	1323	{g,a}	1.4	9	1895	{g,a}	-0.3
{v,g}	0.9	7	1171	{a,g}	0.7	7	956	{a,g}	0.22
{v,v}	1.1	7	1380	{a,a}	0.9	7	1123	{a,a}	0.23

On remarque que les répétitions de buts ne permettent ici aucun gain de précision pour les substitutions abstraites en sortie. Comme dans le cas de quicksort, on constate une amélioration relative des performances de calcul. Dans l'ensemble, la méthode de répétitions de buts s'avère donc efficace dans un certain nombre de cas et peu pénalisante dans les autres. Remarquez également les prédicats "notequ" et "load" ainsi que le foncteur "add". Il s'agissait à l'origine des prédicats "<>" et ":=", et du foncteur "+". Ceux-ci n'étant pas admissibles par la syntaxe d'entrée (prédicats et foncteurs infixés) de l'interpréteur ils ont donc été transformés en des équivalents en notation préfixée.

CONCLUSION

Conclusion.

Tout au long de ce mémoire nous avons progressivement introduit une méthode d'interprétation abstraite par grammaires attribuées. En développant le programme AAIAG nous avons démontré la faisabilité et la fiabilité de la méthode. En guise de conclusion nous voudrions suggérer quelques perspectives d'amélioration de la méthode et de l'interpréteur abstrait.

- Au dernier chapitre nous avons effectué quelques tests comparatifs concernant la répétition de buts. La méthode s'est avérée efficace dans un certain nombre de cas sans être exagérément pénalisante dans les autres. Il serait vraisemblablement profitable de perfectionner la technique utilisée. Les répétitions de buts étaient effectuées statiquement. La définition de critères de réexécution de buts, tant de manière statique que dynamique, permettrait probablement des gains substantiels tant au niveau des temps de calcul qu'au niveau de la précision des résultats (pour les domaines simples du moins). De telles possibilités ont déjà été abordées dans [7].

- Le caractère bipolaire de la méthode, à savoir la coexistence des domaines de l'interprétation abstraite et des grammaires attribuées, constitue un atout; certains résultats antérieurs concernant un de ces domaines pouvant être appliqués à l'ensemble. Ainsi il serait judicieux d'effectuer une implémentation plus générale que celle concrétisée par AAIAG en utilisant un langage pour grammaires attribuées. L'obtention d'un tel programme permettrait une exploration utile de voies d'amélioration telles que celles proposées ici. En particulier certaines techniques de calcul incrémental pour les attributs de la grammaire pourraient être avantageusement utilisées au moyen d'un langage pour grammaires approprié.

- Une manière simple d'améliorer la précision des résultats fournis par l'interpréteur construit serait de définir un certain nombre d'opérateurs abstraits dédiés pour les foncteurs et prédicats fréquemment utilisés tels que ceux des fonctions mathématiques ou encore des prédicats de comparaison ("<","≥",...).

- Diverses techniques d'optimisation comme le caching pour les tables eta ou encore le calcul de points fixes locaux permettraient probablement d'améliorer les performances de l'interpréteur. Le lecteur intéressé trouvera une évaluation pratique de telles techniques dans [6].

- Les aspects relatifs à l'espace mémoire requis par les arbres syntaxiques engendrés par la grammaire n'ont pas été abordés dans le texte présent. L'utilisation d'arbres syntaxiques partiels ou de représentations compactes devraient également être considérée.

Comme on vient de le suggérer l'interprétation abstraite par grammaires attribuées pourrait s'avérer être une méthode payante. Au moment où nous écrivons ces lignes la recherche en la matière se poursuit. Nous osons espérer la concrétisation des espérances que nous y avons placées.

BIBLIOGRAPHIE

Bibliographie.

[1] A. Aho, R. Sethi et J. Ullman. Compilateurs : principes, techniques et outils. InterEditions, ed. 1989.

[2] K. Barbar, and K. Musumbu. Implementation of Abstract Interpretation Algorithms by Means of Attribute-Grammar, IEEE, SSST, March 1994, pages 87-93.

[3] M.M. Corsini, and K. Musumbu. Type inference in Prolog : a new approach. In Theoretical Computer Science n°119 (1993) pages 23-28.

[4] D.E. Knuth. Semantics of Context-Free Languages. In Mathematical Systems theory, vol. 2, n°2. Springer-Verlag Publisher.

[5] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In ACM Transactions on Programming Languages and Systems (TOPLAS), january 94.

[6] B. Le Charlier, O. Degimbe, L. Michel, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms : Practical Efficiency for the Abstract Interpretation of Prolog. In Cousot P. and all, editors, Proc of the Third International Workshop on Static Analysis (WSA '93), number 724 in Lecture Notes in Computer Science, Padova, September 1993. Springer-Verlag.

[7] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. In K. Apt, editor, Proceedings of the Join International Conference and Symposium on Logic Programming (JICSLP '92), Washington, U.S.A., November 1992. MIT Press.

[8] B. Le Charlier. L'Analyse Statique des Programmes par Interprétation Abstraite. Nouvelles de la Science et des Technologies, 9(4):19--25, 1992.

[9] B. Le Charlier and P. Van Hentenryck. On the design of generic abstract interpretation frameworks. In M. Billaud and all, editors, Proceedings of the Workshop on Static Analysis (WSA '92), Bordeaux, France, September 1992. Bigre 81-82.

[10] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, Proceedings of the Eighth International Conference on Logic Programming (ICLP '91), Paris, France, June 1991. MIT Press.

[11] J.W. Lloyd. Foundations of Logic Programming. Second, extended edition, 1987. Spriger-Verlag.

[12] P.Y. Schobbens. Cours de techniques d'Intelligence artificielle, Spécifications et preuves formelles de programmes.

ANNEXE A : SCHEMA DE TRADUCTION LEX & YACC (MODULES TREES.LEX & TREES.YACC)

```
%{

#include <string.h>

extern char buff[256];
extern bool err;

%}

uprcaseltr      [A-Z]
lwrcaseltr      [a-z]
idsymbols       [a-zA-Z0-9_]
digit           [0-9]
blanks          [\ \t]

%%

{blanks}*                   { strcat(buff,yytext); }
\n*                         { if (err) { printf("%s\n",buff); err=FALSE; } strcpy(buff,""); }
{uprcaseltr}{idsymbols}*    { strcat(buff,yytext); copier_string(yylval.string,yytext) return(VAR); }
{digit}+                    { strcat(buff,yytext); copier_string(yylval.string,yytext) return(NPF); }
{lwrcaseltr}{idsymbols}*    { strcat(buff,yytext); copier_string(yylval.string,yytext) return(NPF); }
.                           { strcat(buff,yytext); return(yytext[yyleng-1]); }
```

```
%{

#include "typabs.h"
#include "opabs.h"
#include "trees.h"

extern bool stp;

%}

%union  { char *string;
          lst_var pointlvar;
          ndb *pointb;
          ndsb *pointsb;
          ndsc *pointsc;
          c strc;
          tc strtc;
          fc strfc;
        }

%start sc

%token <string> NPF VAR

%type <strtc> tc
%type <strfc> fc
%type <pointlvar> lvar
%type <pointb> b
%type <pointsb> sb
%type <strc> c
%type <pointsc> sc

%%

sc : c              { if (!stp) { $$=creer_noeud_sc(4,$1.pc,NULL); init_defs($1.npred,$1.lvar,$$); } }
   | c sc           { if (!stp) { $$=creer_noeud_sc(5,$1.pc,$2); maj_defs($1.npred,$1.lvar,$$); } }
c : tc ':' '-' sb '.'  { if (!stp) { $$.npred=$1.npred; $$.lvar=$1.lvar; $$.pc=creer_noeud_c($4); } }
  | tc '.'          { if (!stp) { $$.npred=$1.npred; $$.lvar=$1.lvar;
                      $$.pc=creer_noeud_c(creer_noeud_sb(7,NULL,NULL)); } }
 | error '.'        { yyerrok; }
sb : b              { if (!stp) $$=creer_noeud_sb(8,$1,creer_noeud_sb(7,NULL,NULL)); }
   | b ',' sb       { if (!stp) $$=creer_noeud_sb(8,$1,$3); }
b : VAR '=' VAR     { if (!stp)
                      $$=creer_noeud_b(9,creer_noeud_var($1,creer_noeud_var($3,NULL)),NULL,NULL);
                    }
  | VAR '=' fc      { if (!stp) $$=creer_noeud_b(10,creer_noeud_var($1,$3.lvar),$3.nfonc,NULL); }
  | tc              { if (!stp) $$=creer_noeud_b(11,$1.lvar,NULL,creer_noeud_fi($1.npred)); }
tc : NPF '(' lvar ')' { if (!stp) { $$.npred=$1; $$.lvar=$3; maj_lst_pred($1); } }
fc : NPF            { if (!stp) { $$.lvar=NULL; $$.nfonc=$1; } }
  | NPF '(' lvar ')' { if (!stp) { $$.lvar=$3; $$.nfonc=$1; } }
  | '[' VAR '|' VAR ']' { if (!stp) { $$.lvar=creer_noeud_var($2,creer_noeud_var($4,NULL));
                          $$.nfonc="[|]"; } }
  | '[' ']'         { if (!stp) { $$.lvar=NULL; $$.nfonc="[]"; } }
lvar : VAR          { if (!stp) $$=creer_noeud_var($1,NULL); }
     | VAR ',' lvar { if (!stp) $$=creer_noeud_var($1,$3); }

%%
```

```
#include "treeslex.c"
```

ANNEXE B : MODULE AAIAG.C

```
#include <stdio.h>
#include <string.h>
#include "typabs.h"
#include "opabs.h"
#include "trees.h"
#include "utils.h"
#include "gramabs.h"
#include "view.h"

extern defpred *defs;
extern lst_pred preds;
extern FILE *yyin;
extern bool ext;

unsigned int niter;
subst_abs bin,bout;
char np[128];
bool view;
unsigned int nsp;

void aff_resultats()

 { var_abs *va;

   printf("bin=");
   view_subst(bin);
   printf("\nbout=");
   view_subst(bout);
   printf("\n%d iterations effectuees\n",niter);
   printf("%u sommets parcourus\n",nsp);
 }

void saisie_subst(defpred *dp)

 { var *v;
   char s[8];
   var_abs *va;
   bool ok;

   bin=NULL;
   v=dp->lvar;
   while (v!=NULL)
    { do
       { ok=TRUE;
         printf("%s/",v->nvar);
         scanf("%s",s);
         if (strcmp(s,"var")==0)
          { va=ajout_var_subst(v->nvar,&bin);
            *va->val=VAR;
              }
         else
          if (strcmp(s,"any")==0)
           { va=ajout_var_subst(v->nvar,&bin);
             *va->val=ANY;
               }
          else
           if (strcmp(s,"ground")==0)
            { va=ajout_var_subst(v->nvar,&bin);
```

```
            *va->val=GROUND;
                  }
          else
           { printf("Erreur : valeur non admise.\n");
            ok=FALSE;
                  }
          }
     while (!ok);
     v=v->succ;
    }
 }

defpred *saisie_nom_pred()

 { defpred *dp;

  do
   { printf("Predicat d'appel : ");
    scanf("%s",np);
    dp=rech_def_pred(np);
    if (dp==NULL)
     printf("Erreur : predicat non defini.\n");
   }
  while (dp==NULL);
  return dp;
 }

void iteration(arbre t)

 { subst_abs sa;

  niter++;
  if (view)
   fprintf(stdout,"******** ITERATION %u********\n",niter);
  eval_arbre(t);
  if (t->ichout)
   { bout=copier_subst(t->bout);
    suppr_arbre(t);
   }
  else
   { develop_arbre(t);
    t->etain=t->etaout;
    t->etaout=NULL;
    iteration(t);
   }
 }

void sign_pred_ndef(void)

 { lst_pred lp,t;

  lp=preds;
  while (lp!=NULL)
   { if (rech_def_pred(lp->npred)==NULL)
      printf("Predicat '%s' non defini\n",lp->npred);
     t=lp;
     lp=lp->succ;
     free(t->npred);
     free(t);
```

```
    }
  }

void intabs()

  { arbre t;

    niter=0;
    ext=TRUE;
    nsp=0;
    t=init_arbre(np);
    sign_pred_ndef();
    t->bin=bin;
    t->etain=NULL;
    iteration(t);
  }

void main(int argc,char *argv[])

  { defpred *dp;
    subst_abs sa;

    if (argc<2)
      printf("Fichier Prolog necessaire\n");
    else
      { yyin=fopen(argv[1],"r");
        if (yyin==NULL)
          { printf("Fichier %s introuvable\n",argv[1]);
            exit(1);
          }
        if (argc>2 && strcmp(argv[2],"/v")==0)
          view=TRUE;
        else
          view=FALSE;
        gen_arbres();
        close(yyin);
        dp=saisie_nom_pred();
        saisie_subst(dp);
        printf("\n");
        intabs();
        aff_resultats();
        suppr_subst(bin);
        suppr_subst(bout);
        free(np);
        exit(0);
      }
  }
```

ANNEXE C : MODULE TREES.C

```
#define develop_arbre(a) visite_root(a)
#define copier_arbre(a) copier_sc(a)
#define suppr_arbre(a) { ext=FALSE; visite_root(a); }
#define eval_arbre(a) appl_prod_1(a)
#define init_attr(p) { p->anc=NULL; p->bin=NULL; p->bout=NULL; p->etain=NULL; p-
>etaout=NULL; p->ichin=TRUE; p->ichout=TRUE; }

typedef struct { int prod;
                 tbl_anc anc;
                 subst_abs bin,bout;
                 tbl_eta etain,etaout;
                 void *succ;
                 lst_var lvar;
                 char *nfonc;
                 bool ichin,ichout;
               } ndb;

typedef struct nsb { int prod;
                     tbl_anc anc;
                     subst_abs bin,bout;
                     tbl_eta etain,etaout;
                     ndb *succ1;
                     struct nsb *succ2;
                     bool ichin,ichout;
                        } ndsb;

typedef struct { int prod;
                 tbl_anc anc;
                 subst_abs bin,bout;
                 tbl_eta etain,etaout;
                 ndsb *succ;
                 bool ichin,ichout;
               } ndc;

typedef struct nsc { int prod;
                     tbl_anc anc;
                     subst_abs bin,bout;
                     tbl_eta etain,etaout;
                     ndc *succ1;
                     struct nsc *succ2;
                     bool ichin,ichout;
                   } ndsc;

typedef struct { int prod;
                 tbl_anc anc;
                 subst_abs bin,bout;
                 tbl_eta etain,etaout;
                 ndsc *succ;
                 bool ichin,ichout;
                 char *npred;
                 lst_var lvar;
               } ndfi;

typedef struct { int prod;
                 subst_abs bin,bout;
                 tbl_eta etain,etaout;
                 ndfi *succ;
                 bool ichout;
```

```
                } ndroot;

typedef ndroot * arbre;

typedef struct dp { char *npred;
                    lst_var lvar;
                    ndsc *arbre;
                    struct dp *succ;
                  } defpred;

typedef struct { char *npred;
                 lst_var lvar;
                 ndc *pc;
               } c;

typedef struct { char *npred;
                 lst_var lvar;
               } tc;

typedef struct { char *nfonc;
                 lst_var lvar;
               } fc;

typedef struct lp { char *npred;
                    struct lp *succ;
                  } elem_lst_pred;

typedef elem_lst_pred * lst_pred;

void yyerror(char *);
var *creer_noeud_var(char *,var *);
ndb *creer_noeud_b(int,lst_var,char *,ndfi *);
ndsb *creer_noeud_sb(int,ndb *,ndsb *);
ndc *creer_noeud_c(ndsb *);
ndsc *creer_noeud_sc(int,ndc *,ndsc *);
ndfi *creer_noeud_fi(char *);
void init_defs(char *,lst_var,ndsc *);
void maj_defs(char *,lst_var,ndsc *);
void gen_arbres(void);
defpred *rech_def_pred(char *);
arbre init_arbre(char *);
ndfi *copier_fi(ndfi *);
ndb *copier_b(ndb *);
ndsb *copier_sb(ndsb *);
ndc *copier_c(ndc *);
ndsc *copier_sc(ndsc *);
void visite_b(ndb *);
void visite_sb(int,ndsb *);
void visite_c(ndc *);
void visite_sc(ndsc *);
void visite_fi(int,ndfi *);
void visite_root(ndroot *);
void maj_lst_pred(char *);
```

```
#include <stdio.h>
#include "typabs.h"
#include "opabs.h"
#include "trees.h"
#include "utils.h"

extern int yylineno;
extern int yyparse(void);

defpred *defs;
lst_pred preds;
bool stp,err,ext;
char buff[256];

void yyerror(char *s)

 { fprintf(stderr,"Erreur ligne %d : ",yylineno);
   err=TRUE;
   stp=TRUE;
 }

var *creer_noeud_var(char *nvar,var *succ)

 { var *pvar;

   pvar=(var *)malloc(sizeof(var));
   pvar->nvar=nvar;
   pvar->succ=succ;
   return pvar;
 }

ndb *creer_noeud_b(int prod,lst_var lvar,char *nfonc,ndfi *succ)

 { ndb *pb;

   pb=(ndb *)malloc(sizeof(ndb));
   init_attr(pb);
   pb->prod=prod;
   pb->lvar=lvar;
   pb->nfonc=nfonc;
   pb->succ=succ;
   return pb;
 }

ndsb *creer_noeud_sb(int prod,ndb *succ1,ndsb *succ2)

 { ndsb *psb;

   psb=(ndsb *)malloc(sizeof(ndsb));
   init_attr(psb)
   psb->prod=prod;
   psb->succ1=succ1;
   psb->succ2=succ2;
   return psb;
 }

ndc *creer_noeud_c(ndsb *succ)

 { ndc *pc;
```

```
  pc=(ndc *)malloc(sizeof(ndc));
  init_attr(pc)
  pc->prod=6;
  pc->succ=succ;
  return pc;
 }

ndsc *creer_noeud_sc(int prod,ndc *succ1,ndsc *succ2)

 { ndsc *psc;

  psc=(ndsc *)malloc(sizeof(ndsc));
  init_attr(psc)
  psc->prod=prod;
  psc->succ1=succ1;
  psc->succ2=succ2;
  return psc;
 }

ndfi *creer_noeud_fi(char *npred)

 { ndfi *pfi;

  pfi=(ndfi *)malloc(sizeof(ndfi));
  init_attr(pfi)
  pfi->prod=3;
  pfi->succ=NULL;
  pfi->npred=npred;
  pfi->lvar=NULL;
  return pfi;
 }

void init_defs(char *npred,lst_var lvar,ndsc *arbre)

 { defs=(defpred *)malloc(sizeof(defpred));
  defs->npred=npred;
  defs->lvar=lvar;
  defs->arbre=arbre;
  defs->succ=NULL;
 }

void maj_defs(char *npred,lst_var lvar,ndsc *arbre)

 { defpred *pdp;
  var *pv1,*pv2,*pv;
  bool f;

  pdp=defs;
  while (TRUE)
   if (strcmp(npred,pdp->npred)==0)
        { pv1=lvar;
         pv=pdp->lvar;
         while (pv1!=NULL)
          { if (strcmp(pv->nvar,pv1->nvar)!=0)
                { printf("Erreur : Definition de predicat non homogene (predicat '%s')",npred);
                 exit(1);
                }
           pv2=pv1;
```

```
            pv1=pv1->succ;
            pv=pv->succ;
            free(pv2->nvar);
            free(pv2);
           }
          pdp->arbre=arbre;
          f=TRUE;
          break;
         }
   else
        if (pdp->succ!=NULL)
         pdp=pdp->succ;
        else
         { f=FALSE;
          break;
         }
  if (!f)
   { pdp->succ=(defpred *)malloc(sizeof(defpred));
        pdp=pdp->succ;
        pdp->npred=npred;
        pdp->lvar=lvar;
        pdp->arbre=arbre;
        pdp->succ=NULL;
        arbre->prod=4;
        arbre->succ2=NULL;
   }
 }

void gen_arbres()

 { defs=NULL;
  preds=NULL;
  err=FALSE;
  stp=FALSE;
  strcpy(buff,"");
  yyparse();
  if (stp)
   exit(1);
 }

defpred *rech_def_pred(char *np)

 { defpred *dp;

  dp=defs;
  while (dp!=NULL)
   if (strcmp(np,dp->npred)!=0)
        dp=dp->succ;
   else
        break;
  return dp;
 }

arbre init_arbre(char *np)

 { ndroot *ndr;
  ndfi *ndf;
  defpred *dp;
```

```
  ndr=(ndroot *)malloc(sizeof(ndroot));
  ndr->prod=1;
  ndr->bin=NULL;
  ndr->bout=NULL;
  ndr->etain=NULL;
  ndr->etaout=NULL;
  ndr->ichout=TRUE;
  ndr->succ=ndf=(ndfi *)malloc(sizeof(ndfi));
  ndf->prod=3;
  init_attr(ndf)
  copier_string(ndf->npred,np)
  dp=rech_def_pred(np);
  ndf->lvar=copier_lst_var(dp->lvar);
  ndf->succ=NULL;
  return ndr;
 }

ndfi *copier_fi(ndfi *pnfo)

 { ndfi *pnfd;
  defpred *dp;

  pnfd=(ndfi *)malloc(sizeof(ndfi));
  pnfd->prod=pnfo->prod;
  init_attr(pnfd)
  copier_string(pnfd->npred,pnfo->npred)
  dp=rech_def_pred(pnfd->npred);
  if (dp!=NULL)
   pnfd->lvar=copier_lst_var(dp->lvar);
  else
   pnfd->lvar=NULL;
  pnfd->succ=NULL;
  return pnfd;
 }

ndb *copier_b(ndb *pbo)

 { ndb *pbd;

  pbd=(ndb *)malloc(sizeof(ndb));
  pbd->prod=pbo->prod;
  init_attr(pbd)
  if (pbo->nfonc!=NULL)
   copier_string(pbd->nfonc,pbo->nfonc)
  else
   pbd->nfonc=NULL;
  pbd->lvar=copier_lst_var(pbo->lvar);
  if (pbd->prod==11)
   pbd->succ=copier_fi(pbo->succ);
  else
   pbd->succ=NULL;
  return pbd;
 }

ndsb *copier_sb(ndsb *psbo)

 { ndsb *psbd;

  psbd=(ndsb *)malloc(sizeof(ndsb));
```

```
  psbd->prod=psbo->prod;
  init_attr(psbd);
  if (psbd->prod==8)
    { psbd->succ1=copier_b(psbo->succ1);
        psbd->succ2=copier_sb(psbo->succ2);
    }
  else
    { psbd->succ1=NULL;
        psbd->succ2=NULL;
    }
  return psbd;
 }

ndc *copier_c(ndc *pco)

 { ndc *pcd;

  pcd=(ndc *)malloc(sizeof(ndc));
  pcd->prod=pco->prod;
  init_attr(pcd)
  pcd->succ=copier_sb(pco->succ);
  return pcd;
 }

ndsc *copier_sc(ndsc *psco)

 { ndsc *pscd;

  pscd=(ndsc *)malloc(sizeof(ndsc));
  pscd->prod=psco->prod;
  init_attr(pscd)
  pscd->succ1=copier_c(psco->succ1);
  if (pscd->prod==5)
    pscd->succ2=copier_sc(psco->succ2);
  else
    pscd->succ2=NULL;
  return pscd;
 }

void visite_b(ndb *pnb)

 { if (pnb->prod==11)
    visite_fi(11,pnb->succ);
  pnb->anc=NULL;
  suppr_subst(pnb->bin);
  pnb->bin=NULL;
  pnb->etain=NULL;
  pnb->ichin=TRUE;
  suppr_subst(pnb->bout);
  pnb->bout=NULL;
  pnb->etaout=NULL;
  pnb->ichout=TRUE;
  if (!ext)
    { suppr_lst_var(pnb->lvar);
        free(pnb->nfonc);
        free(pnb);
    }
 }
```

```
void visite_sb(int np,ndsb *pnsb)

 { if (pnsb->prod==8)
    { visite_b(pnsb->succ1);
         visite_sb(8,pnsb->succ2);
    }
   pnsb->anc=NULL;
   if (np==8)
    suppr_subst(pnsb->bin);
   pnsb->bin=NULL;
   pnsb->etain=NULL;
   pnsb->ichin=TRUE;
   pnsb->bout=NULL;
   pnsb->etaout=NULL;
   pnsb->ichout=TRUE;
   if (!ext)
    free(pnsb);
 }

void visite_c(ndc *pnc)

 { visite_sb(6,pnc->succ);
   pnc->anc=NULL;
   pnc->bin=NULL;
   pnc->etain=NULL;
   pnc->ichin=TRUE;
   suppr_subst(pnc->bout);
   pnc->bout=NULL;
   pnc->etaout=NULL;
   pnc->ichout=TRUE;
   if (!ext)
    free(pnc);
 }

void visite_sc(ndsc *pnsc)

 { visite_c(pnsc->succ1);
   if (pnsc->prod==5)
    visite_sc(pnsc->succ2);
   pnsc->anc=NULL;
   pnsc->bin=NULL;
   pnsc->etain=NULL;
   pnsc->ichin=TRUE;
   if (pnsc->prod==5)
    suppr_subst(pnsc->bout);
   pnsc->bout=NULL;
   pnsc->etaout=NULL;
   pnsc->ichout=TRUE;
   if (!ext)
    free(pnsc);
 }

void visite_fi(int np,ndfi *pnf)

 { defpred *dp;
   int p;

   p=pnf->prod;
   if (pnf->prod==2)
```

```
    visite_sc(pnf->succ);
   else
    if (ext)
          if (rech_cpl_dom_anc(pnf->anc,pnf->bin,pnf->npred)==NULL)
           { dp=rech_def_pred(pnf->npred);
            if (dp!=NULL)
          { pnf->succ=copier_arbre(dp->arbre);
              pnf->prod=2;
          }
           }
   if (np==11)
    suppr_subst(pnf->bin);
   pnf->bin=NULL;
   pnf->etain=NULL;
   pnf->ichin=TRUE;
   if (p==3)
    suppr_subst(pnf->bout);
   pnf->bout=NULL;
   pnf->etaout=NULL;
   pnf->ichout=TRUE;
   if (!ext)
    { suppr_lst_var(pnf->lvar);
          free(pnf->npred);
          free(pnf);
    }
 }

void visite_root(ndroot *pnr)

 { visite_fi(1,pnr->succ);
   suppr_tbl_anc(pnr->succ->anc);
   pnr->succ->anc=NULL;
   pnr->etain=NULL;
   pnr->bout=NULL;
   pnr->ichout=TRUE;
   if (!ext)
    { suppr_tbl_eta(pnr->etaout);
          free(pnr);
    }
 }

void maj_lst_pred(char *np)

 { lst_pred lp;
   bool f;

   lp=preds;
   if (lp==NULL)
    { preds=(lst_pred)malloc(sizeof(elem_lst_pred));
      preds->npred=(char *)malloc(strlen(np)+1);
      strcpy(preds->npred,np);
      preds->succ=NULL;
    }
   else
    { f=FALSE;
      do
       { if (strcmp(np,lp->npred)!=0)
          { if (lp->succ!=NULL)
             lp=lp->succ;
```

```
            else
              break;
            }
          else
            f=TRUE;
        }
      while (!f);
      if (!f)
        { lp->succ=(lst_pred)malloc(sizeof(elem_lst_pred));
          lp=lp->succ;
          lp->npred=(char *)malloc(strlen(np)+1);
          strcpy(lp->npred,np);
          lp->succ=NULL;
        }
    }
}
```

ANNEXE D : MODULE OPABS.C

```
#define VARBOTTOM "!"

typedef enum { BOTTOM=1,VAR=2,GROUND=3,ANY=4 } typ_abs;
typedef typ_abs * val_abs;
typedef int tbl_Abi1[5];
```

```
#define bool int
#define TRUE 1
#define FALSE 0
#define copier_val_abs(vn,va) { if (vn==NULL) vn=(val_abs)malloc(sizeof(typ_abs)); *vn=*va; }
#define copier_string(s1,s2) { s1=(char *)malloc(strlen(s2)); strcpy(s1,s2); }
#define test_egal_val_abs(v1,v2) (v1==v2)
#define test_bottom(sa) (*sa->val==BOTTOM)

typedef struct va { char *nvar;
                    val_abs val;
                    struct va *succ;
                       } var_abs;

typedef var_abs * subst_abs;

typedef struct lv { char *nvar;
                    struct lv *succ;
                       } var;

typedef var * lst_var;

typedef struct te { subst_abs bin;
                    char *npred;
                    subst_abs bout;
                    struct te *succ;
                          } trip_eta;

typedef trip_eta * tbl_eta;

typedef struct ta { subst_abs bin;
                    char *npred;
                    struct ta *succ;
                          } cpl_anc;

typedef cpl_anc * tbl_anc;

var_abs *rech_var_subst(char *,subst_abs);
void appl_subst_var(var_abs *,var_abs *,subst_abs);
var_abs *copier_var_abs(var_abs *);
subst_abs copier_subst(subst_abs);
subst_abs Restr_c(subst_abs,subst_abs);
var_abs *ajout_var_subst(char *,subst_abs *);
subst_abs Restr_b(subst_abs,lst_var);
subst_abs Map(lst_var,subst_abs,lst_var);
void suppr_var_abs(var_abs *);
subst_abs suppr_var_subst(char *,subst_abs);
var *rech_var_lst_var(char *,lst_var);
void suppr_subst(subst_abs);
subst_abs Ext_b(subst_abs,subst_abs);
bool test_egal_substs(subst_abs,subst_abs);
trip_eta *rech_cpl_dom_eta(tbl_eta,subst_abs,char *);
tbl_eta ajout_elem_tbl_eta(tbl_eta,subst_abs,char *,subst_abs);
var_abs *creer_bottom(void);
tbl_eta Ext_p(tbl_eta,subst_abs,char *,bool *);
bool comp_val_abs(val_abs,val_abs);
bool comp_subst(subst_abs,subst_abs);
void suppr_trip_eta(trip_eta *);
tbl_eta suppr_elem_tbl_eta(tbl_eta,subst_abs,char *);
tbl_eta Adj(subst_abs,char *,subst_abs,tbl_eta);
```

```
subst_abs Abi1(subst_abs);
subst_abs Abi2(lst_var,subst_abs);
```

```
#include <string.h>
#include "typabs.h"
#include "opabs.h"

tbl_Abi1 tabi1={0,1,2,4,3};

var_abs *rech_var_subst(char *nv,subst_abs sa)

 { var_abs *va;
  int x;

  x=1;
  va=sa;
  while (va!=NULL && x>0)
   { x=strcmp(nv,va->nvar);
    if (x>0)
     va=va->succ;
   }
  if (x!=0)
   va=NULL;
  return va;
 }

void appl_subst_var(var_abs *va1,var_abs *va2,subst_abs sa)

 { va2=rech_var_subst(va2->nvar,sa);
  if (va2!=NULL)
   copier_val_abs(va1->val,va2->val)
  else
   *va1->val=VAR;
 }

var_abs *copier_var_abs(var_abs *vaa)

 { var_abs *van;

  van=(var_abs *)malloc(sizeof(var_abs));
  copier_string(van->nvar,vaa->nvar)
  van->val=NULL;
  copier_val_abs(van->val,vaa->val)
  van->succ=NULL;
  return van;
 }

subst_abs copier_subst(subst_abs sa)

 { var_abs *va,*van1,*van2;
  subst_abs san;

  va=sa;
  san=NULL;
  if (va!=NULL)
   { van2=copier_var_abs(va);
    san=van2;
    va=va->succ;
    while (va!=NULL)
     { van1=copier_var_abs(va);
      van2->succ=van1;
```

```
        van2=van1;
        va=va->succ;
       }
    }
  return san;
 }

var_abs *creer_bottom()

 { var_abs *va;

  va=(var_abs *)malloc(sizeof(var_abs));
  va->nvar=(char *)malloc(2);
  strcpy(va->nvar,VARBOTTOM);
  va->val=(val_abs)malloc(sizeof(typ_abs));
  *va->val=BOTTOM;
  va->succ=NULL;
  return va;
 }

subst_abs Restr_c(subst_abs sa1,subst_abs sa2)

 { var_abs *va;
  subst_abs san;

  if (test_bottom(sa1) || test_bottom(sa2))
   san=creer_bottom();
  else
   { san=copier_subst(sa2);
    va=san;
    while (va!=NULL)
     { appl_subst_var(va,va,sa1);
      va=va->succ;
     }
   }
  return san;
 }

var_abs *ajout_var_subst(char *nv,subst_abs *psa)

 { var_abs *va1,*va2,*van;

  va1=*psa;
  va2=NULL;
  while (va1!=NULL)
   if (strcmp(nv,va1->nvar)>0)
    { va2=va1;
     va1=va1->succ;
    }
   else
    break;
  van=(var_abs *)malloc(sizeof(var_abs));
  copier_string(van->nvar,nv)
  van->val=(val_abs)malloc(sizeof(typ_abs));
  *van->val=VAR;
  van->succ=va1;
  if (va2!=NULL)
   va2->succ=van;
  else
```

```
   *psa=van;
  return van;
 }

subst_abs Restr_b(subst_abs saa,lst_var lv)

 { subst_abs san;
  var_abs *va;

  if (test_bottom(saa))
   san=creer_bottom();
  else
   { san=NULL;
    while (lv!=NULL)
     { va=ajout_var_subst(lv->nvar,&san);
      appl_subst_var(va,va,saa);
      lv=lv->succ;
     }
   }
  return san;
 }

subst_abs Map(lst_var lv1,subst_abs saa,lst_var lv2)

 { var_abs *va1,*va2;
  subst_abs san;

  if (test_bottom(saa))
   san=creer_bottom();
  else
   { san=NULL;
    while (lv2!=NULL)
     { va2=ajout_var_subst(lv2->nvar,&san);
      va1=rech_var_subst(lv1->nvar,saa);
      appl_subst_var(va2,va1,saa);
      lv1=lv1->succ;
      lv2=lv2->succ;
     }
   }
  return san;
 }

void suppr_var_abs(var_abs *va)

 { free(va->nvar);
  free(va->val);
  free(va);
 }

subst_abs suppr_var_subst(char *nv,subst_abs sa)

 { var_abs *va1,*va2;
  int x;

  x=1;
  va1=sa;
  va2=NULL;
  while (va1!=NULL && x>0)
   { x=strcmp(nv,va1->nvar);
```

```
      if (x>0)
        { va2=va1;
          va1=va1->succ;
        }
     }
   if (x==0)
     { if (va2!=NULL)
         va2->succ=va1->succ;
       else
         sa=va1->succ;
       suppr_var_abs(va1);
     }
   return sa;
 }

var *rech_var_lst_var(char *nv,lst_var lv)

 { while (lv!=NULL)
     if (strcmp(lv->nvar,nv)!=0)
       lv=lv->succ;
     else
       break;
   return lv;
 }

void suppr_subst(subst_abs sa)

 { var_abs *va1,*va2;

   va1=sa;
   while (va1!=NULL)
     { va2=va1->succ;
       suppr_var_abs(va1);
       va1=va2;
     }
 }

subst_abs Ext_b(subst_abs saa1,subst_abs saa2)

 { var_abs *va1,*va2;
   subst_abs san;

   if (test_bottom(saa1) || test_bottom(saa2))
     san=creer_bottom();
   else
     { san=copier_subst(saa1);
       va2=saa2;
       while (va2!=NULL)
         { va1=rech_var_subst(va2->nvar,san);
           if (va1==NULL)
             va1=ajout_var_subst(va2->nvar,&san);
           copier_val_abs(va1->val,va2->val)
           va2=va2->succ;
         }
     }
   return san;
 }

bool test_egal_substs(subst_abs sa1,subst_abs sa2)
```

```
{ var_abs *va1,*va2;

  va1=sa1;
  va2=sa2;
  while (va1!=NULL && va2!=NULL)
   if (strcmp(va1->nvar,va2->nvar)==0 && test_egal_val_abs(*va1->val,*va2->val))
        { va1=va1->succ;
      va2=va2->succ;
        }
   else
        break;
  return(va1==NULL && va2==NULL);
 }

trip_eta *rech_cpl_dom_eta(tbl_eta te,subst_abs sa,char *np)

 { trip_eta *tr;

  tr=te;
  while (tr!=NULL)
   { if (strcmp(np,tr->npred)==0)
      if (test_egal_substs(sa,tr->bin))
       break;
     tr=tr->succ;
   }
  return tr;
 }

tbl_eta ajout_elem_tbl_eta(tbl_eta te,subst_abs bin,char *np,subst_abs bout)

 { trip_eta *tr;

  tr=te;
  if (tr!=NULL)
   { while (tr->succ!=NULL)
      tr=tr->succ;
     tr->succ=(trip_eta *)malloc(sizeof(trip_eta));
     tr=tr->succ;
   }
  else
   { tr=(trip_eta *)malloc(sizeof(trip_eta));
     te=tr;
   }
  tr->bin=copier_subst(bin);
  copier_string(tr->npred,np)
  tr->bout=copier_subst(bout);
  tr->succ=NULL;
  return te;
 }

tbl_eta Ext_p(tbl_eta te,subst_abs sa,char *np,bool *ich)

 { subst_abs bout;

   if (rech_cpl_dom_eta(te,sa,np)==NULL)
    { *ich=FALSE;
      bout=creer_bottom();
      te=ajout_elem_tbl_eta(te,sa,np,bout);
```

```
    suppr_subst(bout);
   }
  return te;
 }

bool comp_val_abs(val_abs va1,val_abs va2)

 { bool f;

  if ((*va1==GROUND && *va2==VAR) || (*va1==VAR && *va2==GROUND))
   f=FALSE;
  else
   f=*va1<=*va2;
  return f;
 }

bool comp_subst(subst_abs sa1,subst_abs sa2)

 { var_abs *va1,*va2;
  bool f;

  if (test_bottom(sa1))
   f=TRUE;
  else
   if (test_bottom(sa2))
    f=FALSE;
   else
    { va1=sa1;
     va2=sa2;
     while (va1!=NULL && va2!=NULL)
      { if (strcmp(va1->nvar,va2->nvar)==0 && comp_val_abs(va1->val,va2->val))
         { va1=va1->succ;
          va2=va2->succ;
            }
        else
         break;
       }
     f=va1==NULL && va2==NULL;
    }
  return f;
 }

void suppr_trip_eta(trip_eta *tr)

 { suppr_subst(tr->bin);
  suppr_subst(tr->bout);
  free(tr->npred);
  free(tr);
 }

tbl_eta suppr_elem_tbl_eta(tbl_eta te,subst_abs sa,char *np)

 { trip_eta *tr1,*tr2;

  tr1=te;
  tr2=NULL;
  while (tr1!=NULL)
   { if (strcmp(tr1->npred,np)==0)
      if (test_egal_substs(sa,tr1->bin))
```

```
        { if (tr2!=NULL)
            tr2->succ=tr1->succ;
          else
            te=tr1->succ;
          suppr_trip_eta(tr1);
          break;
             }
      tr2=tr1;
      tr1=tr1->succ;
     }
   return te;
 }

tbl_eta Adj(subst_abs bin,char *np,subst_abs bout,tbl_eta te)

 { trip_eta *tr;

   tr=rech_cpl_dom_eta(te,bin,np);
   if (!comp_subst(bout,tr->bout))
     { te=suppr_elem_tbl_eta(te,bin,np);
       te=ajout_elem_tbl_eta(te,bin,np,bout);
     }
   return te;
 }

subst_abs Abi1(subst_abs saa)

 { subst_abs san;
   var_abs *va1,*va2;

   if (test_bottom(saa))
     san=creer_bottom();
   else
     { san=copier_subst(saa);
       va1=san;
       va2=va1->succ;
       if (tabi1[*va1->val]<tabi1[*va2->val])
         copier_val_abs(va1->val,va2->val)
       else
         copier_val_abs(va2->val,va1->val)
     }
   return san;
 }

subst_abs Abi2(lst_var lv,subst_abs saa)

 { var *v;
   subst_abs san;
   var_abs *va1,*va2,*va;
   bool f1,f2;

   if (test_bottom(saa))
     san=creer_bottom();
   else
     { f1=TRUE;
       v=lv;
       va=rech_var_subst(v->nvar,saa);
       if (!test_egal_val_abs(*va->val,GROUND))
         f1=FALSE;
```

```
    f2=TRUE;
    san=NULL;
    v=v->succ;
    while (v!=NULL)
          { va1=rech_var_subst(v->nvar,saa);
      if (!test_egal_val_abs(*va1->val,GROUND))
       f2=FALSE;
      va2=ajout_var_subst(v->nvar,&san);
      if (f1)
       *va2->val=GROUND;
      else
       copier_val_abs(va2->val,va1->val)
      v=v->succ;
     }
    v=lv;
    va=ajout_var_subst(v->nvar,&san);
    if (f1 || f2)
     *va->val=GROUND;
    else
     *va->val=ANY;
  }
 return san;
}
```

ANNEXE E : MODULE GRAMABS.C

```
void appl_prod_1(ndroot *);
void appl_prod_2(ndfi *);
void appl_prod_3(ndfi *);
void appl_prod_4(ndsc *);
void appl_prod_5(ndsc *);
void appl_prod_6(ndc *);
void appl_prod_7(ndsb *);
void appl_prod_8(ndsb *);
void appl_prod_9(ndb *);
void appl_prod_10(ndb *);
void appl_prod_11(ndb *);

extern bool view;
```

```
#include <stdio.h>
#include "typabs.h"
#include "opabs.h"
#include "trees.h"
#include "utils.h"
#include "gramabs.h"
#include "view.h"

extern unsigned int nsp;

void appl_prod_11(ndb *pnb)

 { ndfi *pnf;

   nsp++;
   pnf=(ndfi *)pnb->succ;
   pnf->anc=pnb->anc;
   pnf->bin=Map(pnb->lvar,pnb->bin,pnf->lvar);
   pnf->ichin=pnb->ichin;
   pnf->etain=Ext_p(pnb->etain,pnf->bin,pnf->npred,&pnf->ichin);
   if (pnf->prod==2)
     appl_prod_2(pnf);
   else
     appl_prod_3(pnf);
   pnb->bout=Map(pnf->lvar,pnf->bout,pnb->lvar);
   pnb->etaout=pnf->etaout;
   pnb->ichout=pnf->ichout;
   if (view)
     view_b(pnb);
 }

void appl_prod_10(ndb *pnb)

 { nsp++;
   pnb->bout=Abi2(pnb->lvar,pnb->bin);
   pnb->etaout=pnb->etain;
   pnb->ichout=pnb->ichin;
   if (view)
     view_b(pnb);
 }

void appl_prod_9(ndb *pnb)

 { nsp++;
   pnb->bout=Abi1(pnb->bin);
   pnb->etaout=pnb->etain;
   pnb->ichout=pnb->ichin;
   if (view)
     view_b(pnb);
 }

void appl_prod_8(ndsb *pnsb1)

 { ndb *pnb;
   ndsb *pnsb2;

   nsp++;
   pnb=pnsb1->succ1;
```

```
  pnsb2=pnsb1->succ2;
  pnb->anc=pnsb1->anc;
  pnb->bin=Restr_b(pnsb1->bin,pnb->lvar);
  pnb->etain=pnsb1->etain;
  pnb->ichin=pnsb1->ichin;
  if (pnb->prod==9)
    appl_prod_9(pnb);
  else
    if (pnb->prod==10)
      appl_prod_10(pnb);
    else
      appl_prod_11(pnb);
  pnsb2->anc=pnsb1->anc;
  /*pnsb2->bin=Ext_b(pnb->lvar,pnsb1->bin,pnb->bout);*/
  pnsb2->bin=Ext_b(pnsb1->bin,pnb->bout);
  pnsb2->etain=pnb->etaout;
  pnsb2->ichin=pnb->ichout;
  if (pnsb2->prod==7)
    appl_prod_7(pnsb2);
  else
    appl_prod_8(pnsb2);
  pnsb1->bout=pnsb2->bout;
  pnsb1->etaout=pnsb2->etaout;
  pnsb1->ichout=pnsb2->ichout;
  if (view)
    view_sb(pnsb1);
 }

void appl_prod_7(ndsb *pnsb)

 { nsp++;
  pnsb->bout=pnsb->bin;
  pnsb->etaout=pnsb->etain;
  pnsb->ichout=pnsb->ichin;
  if (view)
    view_sb(pnsb);
 }

void appl_prod_6(ndc *pnc)

 { ndsb *pnsb;

  nsp++;
  pnsb=pnc->succ;
  pnsb->anc=pnc->anc;
  pnsb->bin=pnc->bin;
  pnsb->etain=pnc->etain;
  pnsb->ichin=pnc->ichin;
  if (pnsb->prod==7)
    appl_prod_7(pnsb);
  else
    appl_prod_8(pnsb);
  pnc->bout=Restr_c(pnsb->bout,pnc->bin);
  pnc->etaout=pnsb->etaout;
  pnc->ichout=pnsb->ichout;
  if (view)
    view_c(pnc);
 }
```

```
void appl_prod_5(ndsc *pnsc1)

 { ndc *pnc;
   ndsc *pnsc2;

   nsp++;
   pnc=pnsc1->succ1;
   pnsc2=pnsc1->succ2;
   pnc->anc=pnsc1->anc;
   pnc->bin=pnsc1->bin;
   pnc->etain=pnsc1->etain;
   pnc->ichin=pnsc1->ichin;
   appl_prod_6(pnc);
   pnsc2->anc=pnsc1->anc;
   pnsc2->bin=pnsc1->bin;
   pnsc2->etain=pnc->etaout;
   pnsc2->ichin=pnsc1->ichin;
   if (pnsc2->prod==4)
     appl_prod_4(pnsc2);
   else
     appl_prod_5(pnsc2);
   pnsc1->bout=union_subst(pnc->bout,pnsc2->bout);
   pnsc1->etaout=pnsc2->etaout;
   pnsc1->ichout=pnsc2->ichout && pnc->ichout;
   if (view)
     view_sc(pnsc1);
 }

void appl_prod_4(ndsc *pnsc)

 { ndc *pnc;

   nsp++;
   pnc=pnsc->succ1;
   pnc->anc=pnsc->anc;
   pnc->bin=pnsc->bin;
   pnc->etain=pnsc->etain;
   pnc->ichin=pnsc->ichin;
   appl_prod_6(pnc);
   pnsc->bout=pnc->bout;
   pnsc->etaout=pnc->etaout;
   pnsc->ichout=pnc->ichout;
   if (view)
     view_sc(pnsc);
 }

void appl_prod_3(ndfi *pnf)

 { trip_eta *tr;
   subst_abs sa;

   nsp++;
   tr=rech_cpl_dom_eta(pnf->etain,pnf->bin,pnf->npred);
   if (tr!=NULL)
     { pnf->bout=copier_subst(tr->bout);
       /*pnf->etaout=pnf->etain;*/
       pnf->ichout=pnf->ichin;
     }
   else
```

```
    { pnf->bout=creer_bottom();
      /*sa=pnf->bout;
      pnf->etaout=ajout_elem_tbl_eta(pnf->etain,pnf->bin,pnf->npred,sa);*/
      pnf->ichout=FALSE;
    }
  pnf->etaout=pnf->etain;
  if (view)
    view_fi(pnf);
 }

void appl_prod_2(ndfi *pnf)

 { ndsc *pnsc;
  trip_eta *tr;

  nsp++;
  pnsc=pnf->succ;
  pnsc->anc=ajout_elem_tbl_anc(pnf->anc,pnf->bin,pnf->npred);
  pnsc->bin=pnf->bin;
  pnsc->etain=pnf->etain;
  pnsc->ichin=pnf->ichin;
  if (pnsc->prod==4)
    appl_prod_4(pnsc);
  else
    appl_prod_5(pnsc);
  pnf->anc=pnsc->anc;
  pnf->bout=pnsc->bout;
  tr=rech_cpl_dom_eta(pnsc->etaout,pnf->bin,pnf->npred);
  if (!comp_subst(pnsc->bout,tr->bout))
    { pnf->etaout=Adj(pnf->bin,pnf->npred,pnsc->bout,pnsc->etaout);
      pnf->ichout=FALSE;
    }
  else
    { pnf->etaout=pnsc->etaout;
      pnf->ichout=pnsc->ichout;
    }
  if (view)
    view_fi(pnf);
 }

void appl_prod_1(ndroot *pnr)

 { ndfi *pnf;

  nsp++;
  pnf=pnr->succ;
  pnf->anc=NULL;
  pnf->bin=pnr->bin;
  pnf->ichin=TRUE;
  pnf->etain=Ext_p(pnr->etain,pnr->bin,pnf->npred,&pnf->ichin);
  if (pnf->prod==2)
    appl_prod_2(pnf);
  else
    appl_prod_3(pnf);
  pnr->bout=pnf->bout;
  pnr->etaout=pnf->etaout;
  pnr->ichout=pnf->ichout;
  if (view)
    view_root(pnr);
```

```
}
```

ANNEXE F : MODULE UTILS.C

```
tbl_anc ajout_elem_tbl_anc(tbl_anc,subst_abs,char *);
subst_abs union_subst(subst_abs,subst_abs);
cpl_anc *rech_cpl_dom_anc(tbl_anc,subst_abs,char *);
void suppr_tbl_eta(tbl_eta);
void suppr_tbl_anc(tbl_anc);
lst_var copier_lst_var(lst_var);
void suppr_lst_var(lst_var);
```

```
#include <string.h>
#include "typabs.h"
#include "opabs.h"
#include "utils.h"

tbl_anc ajout_elem_tbl_anc(tbl_anc ta,subst_abs sa,char *np)

 { cpl_anc *ca;

  ca=ta;
  if (ca!=NULL)
   { while (ca->succ!=NULL)
      ca=ca->succ;
    ca->succ=(cpl_anc *)malloc(sizeof(cpl_anc));
    ca=ca->succ;
   }
  else
   { ca=(cpl_anc *)malloc(sizeof(cpl_anc));
    ta=ca;
   }
  ca->bin=copier_subst(sa);
  copier_string(ca->npred,np)
  ca->succ=NULL;
  return ta;
 }

subst_abs union_subst(subst_abs sa1,subst_abs sa2)

 { subst_abs san;
  var_abs *va1,*va2;

  if (test_bottom(sa1))
   san=copier_subst(sa2);
  else
   if (test_bottom(sa2))
    san=copier_subst(sa1);
   else
    { san=copier_subst(sa2);
     va1=sa1;
     while (va1!=NULL)
      { va2=rech_var_subst(va1->nvar,san);
          if (va2!=NULL)
           { if ((*va1->val==VAR && *va2->val==GROUND) || (*va1->val==GROUND && *va2-
>val==VAR))
            *va2->val=ANY;
           else
            if (*va2->val<*va1->val)
             copier_val_abs(va2->val,va1->val)
            }
        else
         { va2=ajout_var_subst(va1->nvar,&san);
          copier_val_abs(va2->val,va1->val)
           }
        va1=va1->succ;
       }
    }
  return san;
 }
```

```
cpl_anc *rech_cpl_dom_anc(tbl_anc ta,subst_abs sa,char *np)

 { cpl_anc *ca;

  ca=ta;
  while (ca!=NULL)
   { if (strcmp(np,ca->npred)==0)
      if (test_egal_substs(sa,ca->bin))
       break;
     ca=ca->succ;
   }
  return ca;
 }

void suppr_tbl_eta(tbl_eta te)

 { trip_eta *tr1,*tr2;

  tr1=te;
  while (tr1!=NULL)
   { tr2=tr1->succ;
    suppr_trip_eta(tr1);
    tr1=tr2;
   }
 }

void suppr_tbl_anc(tbl_anc ta)

 { cpl_anc *ca1,*ca2;

  ca1=ta;
  while (ca1!=NULL)
   { ca2=ca1->succ;
    suppr_subst(ca1->bin);
    free(ca1->npred);
    free(ca1);
    ca1=ca2;
   }
 }

lst_var copier_lst_var(lst_var lv)

 { var *vn1,*vn2,*v;
  lst_var lvn;

  v=lv;
  if (v!=NULL)
   { vn2=(var *)malloc(sizeof(var));
    copier_string(vn2->nvar,v->nvar)
    vn2->succ=NULL;
    lvn=vn2;
    v=v->succ;
    while (v!=NULL)
     { vn1=(var *)malloc(sizeof(var));
      copier_string(vn1->nvar,v->nvar)
      vn1->succ=NULL;
      vn2->succ=vn1;
      vn2=vn1;
```

```
        v=v->succ;
       }
    }
  return lvn;
 }

void suppr_lst_var(lst_var lv)

 { var *v1,*v2;

  v1=lv;
  while (v1!=NULL)
   { v2=v1->succ;
     free(v1->nvar);
     free(v1);
     v1=v2;
    }
 }
```

ANNEXE G : MODULE VIEW.C

```
#define view_bool(b) if (b==FALSE) printf("FALSE"); else printf("TRUE")
#define view_var(v) printf("%s",v->nvar)
#define view_at(nd) view_attr(nd->anc,nd->bin,nd->bout,nd->etaout,nd->ichin,nd->ichout)

void view_lst_var(lst_var);
void view_val_abs(val_abs);
void view_var_abs(var_abs *);
void view_subst(subst_abs);
void view_trip_eta(trip_eta *);
void view_tbl_eta(tbl_eta);
void view_cpl_anc(cpl_anc *);
void view_tbl_anc(tbl_anc);
void view_attr(tbl_anc,subst_abs,subst_abs,tbl_eta,bool,bool);
void view_root(ndroot *);
void view_fi(ndfi *);
void view_sc(ndsc *);
void view_c(ndc *);
void view_sb(ndsb *);
void view_b(ndb *);
```

```
#include <stdio.h>
#include <string.h>
#include "typabs.h"
#include "opabs.h"
#include "trees.h"
#include "view.h"

void view_lst_var(lst_var lv)

 { printf("{");
  if (lv!=NULL)
    { view_var(lv);
      lv=lv->succ;
      while (lv!=NULL)
            { printf(",");
          view_var(lv);
          lv=lv->succ;
            }
    }
  printf("}");
 }

void view_val_abs(val_abs v)

 { switch (*v)
    { case VAR : printf("var"); break;
      case ANY : printf("any"); break;
      case GROUND : printf("ground"); break;
      case BOTTOM : printf("bottom");
    }
 }

void view_var_abs(var_abs *va)

 { printf("%s/",va->nvar);
  view_val_abs(va->val);
 }

void view_subst(subst_abs sa)

 { var_abs *va;

  printf("{");
  va=sa;
  if (va!=NULL)
    { view_var_abs(va);
      va=va->succ;
      while (va!=NULL)
            { printf(",");
          view_var_abs(va);
          va=va->succ;
            }
    }
  printf("}");
 }

void view_trip_eta(trip_eta *tr)
```

```
{ printf("bin=");
  view_subst(tr->bin);
  printf("\npred : %s",tr->npred);
  printf("\nbout=");
  view_subst(tr->bout);
}

void view_tbl_eta(tbl_eta te)

{ trip_eta *tr;

  tr=te;
  while (tr!=NULL)
    { view_trip_eta(tr);
      printf("\n");
      tr=tr->succ;
    }
}

void view_cpl_anc(cpl_anc *ca)

{ printf("bin=");
  view_subst(ca->bin);
  printf("\npred : %s",ca->npred);
}

void view_tbl_anc(tbl_anc ta)

{ cpl_anc *ca;

  ca=ta;
  while (ca!=NULL)
    { view_cpl_anc(ca);
      printf("\n");
      ca=ca->succ;
    }
}

void view_attr(tbl_anc ta,subst_abs bin,subst_abs bout,tbl_eta etaout,bool ichin,bool ichout)

{ printf("* anc :\n");
  view_tbl_anc(ta);
  printf("\n* fin anc\n* bin=");
  view_subst(bin);
  printf("\n* bout=");
  view_subst(bout);
  printf("\n* etaout :\n");
  view_tbl_eta(etaout);
  printf("\n* fin etaout\n* ichin=");
  view_bool(ichin);
  printf("\n* ichout=");
  view_bool(ichout);
}

void view_root(ndroot *pnr)

{ printf("*** Root : ***\n* bin=");
  view_subst(pnr->bin);
  printf("\n* bout=");
```

```
  view_subst(pnr->bout);
  printf("\n* etaout :\n");
  view_tbl_eta(pnr->etaout);
  printf("\n* fin etaout\n* ichout=");
  view_bool(pnr->ichout);
  printf("\n*** fin root ***\n");
 }

void view_fi(ndfi *pnf)

 { printf("*** fi : ***\n");
  view_at(pnf);
  printf("\n* pred : %s\n* lvar=",pnf->npred);
  view_lst_var(pnf->lvar);
  printf("\n*** fin fi ***\n");
 }

void view_sc(ndsc *pnsc)

 { printf("*** sc : ***\n");
  view_at(pnsc);
  printf("\n*** fin sc ***\n");
 }

void view_c(ndc *pnc)

 { printf("*** c : ***\n");
  view_at(pnc);
  printf("\n*** fin c ***\n");
 }

void view_sb(ndsb *pnsb)

 { printf("*** sb : ***\n");
  view_at(pnsb);
  printf("\n*** fin sb ***\n");
 }

void view_b(ndb *pnb)

 { printf("*** b : ***\n");
  view_at(pnb);
  printf("\n* fonc : %s\n* lvar=",pnb->nfonc);
  view_lst_var(pnb->lvar);
  printf("\n*** fin b ***\n");
 }
```

ANNEXE H : SCHEMA DE TRADUCTION NP

<SC>::=<C><SC> | <C>

<C>::=<TCAP>:- { produire(tcap.val),":-",tcap.rpt); sb.buff←""; si tcap.rpt≠"" alors produire(","); sb.buff←concat(sb.buff,tcap.rpt) finsi } <SB>. { produire(".") }
| <C> { produire(tcap.val); si tcap.rpt≠"" alors produire(":-",tcap.rpt,".") finsi }

<SB_1>::=<B>, { si b.rpt≠"" alors produire(b.rpt) finsi; produire(b.val,","); si b.tcap alors si sb_1.buff≠"" alors produire(sb_1.buff,",") finsi; si b.rpt≠"" alors produire(b.rpt,",") finsi; sb_2.buff←"" sinon si sb_1.buff≠"" alors sb_2.buff←concat(sb_1.buff,",") sinon sb_2.buff←"" finsi; si b.rpt≠"" alors sb_2.buff←concat(sb_2.buff,b.rpt,",") finsi; sb_2.buff←concat(sb_2.buff,b.val) } <SB_2>
| <B> { si b.rpt≠"" alors produire(b.rpt,",") finsi; produire(b.val); si b.tcap alors si sb_1.buff≠"" alors produire(",",sb_1.buff) finsi; si b.rpt≠"" alors produire(",",b.rpt) finsi; finsi }

<B>::=<VAR_1>=<VAR_2>	{ b.val←concat(var_1.val,"=",var_2.val); b.rpt←""; b.tcap←faux }
| <VAR>=<FC>	{ b.val←concat(var.val,"=",fc.val); b.rpt←fc.rpt; b.tcap←faux }
| <TCAP>	{ b.val←tcap.val; b.rpt←tcap.rpt; b.tcap←vrai }
<FC>::=<NPF>(<LARG>)	{ fc.val←concat(npf.val,"(",larg.val,")"); fc.rpt←larg.rpt }
| <NPF>	{ fc.val←npf.val; fc.rpt←"" }
| [<VAR_1>|<VAR_2>]	{ fc.val←concat("[",var_1.val,"|",var_2.val,"]"); fc.rpt←"" }
| []	{ fc.val←"[]"; fc.rpt←"" }
| <CST>	{ fc.val←cst.val; cst.rpt←"" }
<TCAP>::=<NPF>(<LARG>)	{ tcap.val←concat(npf.val,"(",larg.val,")"); tcap.rpt←larg.rpt }
| <NPF>	{ fc.val←npf.val; fc.rpt←"" }
<$LARG_1$>::=<ARG>,<$LARG_2$>|	{ $larg_1$.val←concat(arg.val,",",$larg_2$.val); si arg.rpt≠"" alors si arg.rpt≠"" alors si $larg_2$.rpt≠"" alors $larg_1$.rpt←concat(arg.rpt,",",$larg_2$.rpt) sinon $larg_1$.rpt←arg.rpt finsi sinon $larg_1$.rpt←$larg_2$.rpt finsi }
| <NPF>	{ tcap.val←npf.val; tcap.rpt←"" }
<ARG>::=<VAR>	{ arg.val←var.val; arg.rpt←"" }
| [<VAR_1>|<VAR_2>]	{ arg.val←newvar; arg.rpt←concat(arg.val,"=[",var_1.val,"|",var_2.val,"]") }
| []	{ arg.val←newvar; arg.rpt←concat(arg.val,"=[]") }
| <NPF>(<LARG>)	{ arg.val←newvar; si larg.rpt≠"" alors arg.rpt←concat(larg.rpt,",",npf.val,"(",larg.val,")") sinon arg.rpt←concat(npf.val,"(",larg.val,")") finsi }
| <NPF>	{ arg.val←newvar; arg.rpt←concat(arg.val,"=",npf.val) }
| <CST>	{ arg.val←newvar; arg.rpt←concat(arg.val,"=",cst.val) }

ANNEXE I : MODULE NP.C

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define Boolean int

#define TRUE 1
#define FALSE 0
#define TSTR 512
#define DSTR 2048
#define LTAB 5000
#define FDC '\0'
#define FDL '\n'
#define TAB '\t'
#define FDF 0
#define VAR 256
#define NPF 257
#define CST 258

typedef struct { unsigned int lex;
                 char val[TSTR];
               } SymbTerm;

typedef struct { char val[TSTR],rpt[TSTR],buff[DSTR];
                 Boolean tcap;
               } SymbNTerm;

typedef struct v { char *nvar;
                   struct v *sv;
                 } Var;

void Erreur(void);
void AnalLex(void);
void Accepter(unsigned int);
void IToA(int,char []);
char *NewVar(void);
void SymbArg(void);
void SymbLArg(void);
void SymbTCAP(void);
void SymbFc(void);
void SymbB(void);
void SymbSB(void);
void SymbC(void);
void SymbSC(void);
unsigned int ChargTxt(char *[]);
void NomPred(char *,char []);
void TriPred(char *[],unsigned int);
struct v *ConstrLstVar(char *);
unsigned int RechNomVar(char *,char *,unsigned int);
char *UnifCls(struct v *,char *);
void LstNewVar(struct v*);
void UnifNomVars(char *[],unsigned int);
void SauverTxt(char *[],unsigned int);
void NormPred(void);
void main(int,char *[]);
```

```
FILE *fin,*fout;
unsigned int nlg;
SymbTerm sterm;
SymbNTerm snterm;
char lc[DSTR];
Boolean buf;

void Erreur()

 { char s[TSTR];
   int c;
   unsigned int i;

   i=1;
   c=fgetc(fin);
   while (c!=EOF && c!='.')
    { if (!isspace(c))
            { s[i]=c;
              i++;
            }
          c=getc(fin);
    }
   if (c=='.')
    { s[i]='.';
          i++;
    }
   s[0]='^';
   s[i]=FDC;
   strcat(lc,s);
   printf("Erreur ligne %u : %s\n",nlg,lc);
   fclose(fin);
   fclose(fout);
   exit(1);
 }

void AnalLex()

 { int c;
   unsigned int i;
   char *s;

   while (TRUE)
    { c=getc(fin);
          if (isalpha(c))
           { if (isupper(c))
              sterm.lex=VAR;
             else
              sterm.lex=NPF;
             sterm.val[0]=c;
             c=getc(fin);
             i=1;
             while (isalnum(c))
              { sterm.val[i]=c;
                    c=getc(fin);
                    i++;
              }
```

```
            sterm.val[i]=FDC;
            if (c!=EOF)
              ungetc(c,fin);
            strcat(lc,sterm.val);
            break;
           }
          else
           if (isdigit(c))
            { sterm.lex=CST;
             sterm.val[0]=c;
             c=getc(fin);
             i=1;
             while (isdigit(c))
                  { sterm.val[i]=c;
                   c=getc(fin);
                   i++;
                  }
             sterm.val[i]=FDC;
             if (c!=EOF)
                  ungetc(c,fin);
             strcat(lc,sterm.val);
             break;
            }
           else
            if (c==EOF)
             { sterm.lex=FDF;
                  break;
             }
            else
             if (c==FDL)
                  { nlg++;
                   strcpy(lc,"");
                  }
             else
                  if (!isspace(c))
                   { sterm.lex=c;
             s=(char *)malloc(2);
             s[0]=c;
             s[1]=FDC;
                    strcat(lc,s);
             free(s);
                    break;
                   }
    }
}

void Accepter(unsigned int s)

{ if (s==sterm.lex)
    AnalLex();
  else
    Erreur();
}

void IToA(int n,char nv[])

{ unsigned i,j,r;
```

```
  char nw[6];

  i=4;
  do
   { r=n % 10;
        nw[i]='0'+r;
        i--;
        n=(n-r)/10;
   }
  while (n>0);
  nw[5]=FDC;
  i++;
  j=0;
  while (i<6)
   { nv[j]=nw[i];
        i++;
        j++;
   }
 }

char *NewVar()

 { static int n=0;
  char *v;
  char nv[6];

  strcpy(nv,"");
  IToA(n,nv);
  v=(char *)malloc(8);
  strcpy(v,"YY");
  strcat(v,nv);
  n++;
  return v;
 }

void SymbArg()

 { char tmp1[TSTR],tmp2[TSTR];

  switch (sterm.lex)
   { case VAR: strcpy(snterm.val,sterm.val);
                strcpy(snterm.rpt,"");
                Accepter(VAR);
                break;
       case '[': Accepter('[');
                switch (sterm.lex)
                 { case VAR: strcpy(snterm.val,NewVar());
                             strcpy(snterm.rpt,snterm.val);
                             strcat(snterm.rpt,"=[");
                             strcat(snterm.rpt,sterm.val);
                             Accepter(VAR);
                             strcat(snterm.rpt,"|");
                             Accepter('|');
                             strcat(snterm.rpt,sterm.val);
                             Accepter(VAR);
                             strcat(snterm.rpt,"]");
                             Accepter(']');
```

```
                                    break;
                        case ']': strcpy(snterm.val,NewVar());
                                    strcpy(snterm.rpt,snterm.val);
                                    strcat(snterm.rpt,"=[]");
                                    Accepter(']');
                                    break;
                        default : Erreur();
                        }
                    break;
        case NPF: strcpy(tmp1,NewVar());
                    strcpy(tmp2,sterm.val);
                    Accepter(NPF);
                    if (sterm.lex=='(')
                     { Accepter('(');
                       SymbLArg();
                       if (strcmp(snterm.rpt,"")!=0)
                            strcat(snterm.rpt,",");
                       strcat(snterm.rpt,tmp1);
                       strcat(snterm.rpt,"=");
                       strcat(snterm.rpt,tmp2);
                       strcat(snterm.rpt,"(");
                       strcat(snterm.rpt,snterm.val);
                       strcat(snterm.rpt,")");
                       Accepter(')');
                     }
                    else
                     { strcpy(snterm.rpt,tmp1);
                       strcat(snterm.rpt,"=");
                       strcat(snterm.rpt,tmp2);
                     }
                    strcpy(snterm.val,tmp1);
                    break;
        case CST: Accepter(CST);
                    strcpy(snterm.val,NewVar());
                    strcpy(snterm.rpt,snterm.val);
                    strcat(snterm.rpt,"=");
                    strcat(snterm.rpt,sterm.val);
                    break;
        default : Erreur();
    }
}

void SymbLArg()

{ char val[TSTR],rpt[TSTR];

  SymbArg();
  if (sterm.lex==',')
    { strcpy(val,snterm.val);
        strcpy(rpt,snterm.rpt);
        Accepter(',');
        SymbLArg();
        if (strcmp(rpt,"")!=0)
          { if (strcmp(snterm.rpt,"")!=0)
              { strcat(rpt,",");
                  strcat(rpt,snterm.rpt);
              }
```

```
            }
          else
            strcpy(rpt,snterm.rpt);
          strcat(val,",");
          strcat(val,snterm.val);
          strcpy(snterm.val,val);
          strcpy(snterm.rpt,rpt);
    }
 }

void SymbTCAP()

 { char tmp[TSTR];

   strcpy(tmp,sterm.val);
   Accepter(NPF);
   if (sterm.lex=='(')
    { strcat(tmp,"(");
          Accepter('(');
          SymbLArg();
          strcat(tmp,snterm.val);
          strcat(tmp,")");
          Accepter(')');
    }
   strcpy(snterm.val,tmp);
 }

void SymbFc()

 { char tmp[TSTR];

   switch (sterm.lex)
    { case NPF: strcpy(tmp,sterm.val);
                Accepter(NPF);
                if (sterm.lex=='(')
                  { strcat(tmp,"(");
                    Accepter('(');
                    SymbLArg();
                    strcat(tmp,snterm.val);
                    strcat(tmp,")");
                    Accepter(')');
                  }
                else
                  strcpy(snterm.rpt,"");
                strcpy(snterm.val,tmp);
                break;
          case '[': Accepter('[');
                switch (sterm.lex)
                  { case VAR: strcpy(snterm.val,"[");
                              strcat(snterm.val,sterm.val);
                              Accepter(VAR);
                              strcat(snterm.val,"|");
                              Accepter('|');
                              strcat(snterm.val,sterm.val);
                              Accepter(VAR);
                              strcat(snterm.val,"]");
                              Accepter(']');
```

```
                                strcpy(snterm.rpt,"");
                                break;
                        case ']': strcpy(snterm.val,"[]");
                                strcpy(snterm.rpt,"");
                                Accepter(']');
                                break;
                        default : Erreur();
                        }
                    break;
            case CST: strcpy(snterm.val,sterm.val);
                    strcpy(snterm.rpt,"");
                Accepter(CST);
                break;
            default : Erreur();
    }
}

void SymbB()

{ char val[TSTR];

  switch (sterm.lex)
    { case VAR: strcpy(val,sterm.val);
                    Accepter(VAR);
                    strcat(val,"=");
                    Accepter('=');
                    switch (sterm.lex)
                      { case VAR: strcpy(snterm.val,val);
                                strcat(snterm.val,sterm.val);
                                Accepter(VAR);
                                strcpy(snterm.rpt,"");
                                break;
                        case NPF:
                        case CST:
                        case '[': SymbFc();
                                strcat(val,snterm.val);
                                strcpy(snterm.val,val);
                                break;
                        default : Erreur();
                        }
                    snterm.tcap=FALSE;
                    break;
            case NPF: SymbTCAP();
                    snterm.tcap=TRUE;
                    break;
            default : Erreur();
    }
}

void SymbSB()

{ SymbB();
  while (sterm.lex==',')
    { Accepter(',');
          if (strcmp(snterm.rpt,"")!=0)
            fprintf(fout,"%s,",snterm.rpt);
          fprintf(fout,"%s,",snterm.val);
```

```
        if (snterm.tcap)
         { if (buf && strcmp(snterm.buff,"")!=0)
            fprintf(fout,"%s,",snterm.buff);
           if (buf && strcmp(snterm.rpt,"")!=0)
            fprintf(fout,"%s,",snterm.rpt);
           strcpy(snterm.buff,"");
         }
        else
         { if (strcmp(snterm.buff,"")!=0)
            strcat(snterm.buff,",");
           if (strcmp(snterm.rpt,"")!=0)
            { strcpy(snterm.buff,snterm.rpt);
                 strcat(snterm.buff,",");
            }
           strcat(snterm.buff,snterm.val);
         }
        SymbB();
   }
  if (strcmp(snterm.rpt,"")!=0)
   fprintf(fout,"%s,",snterm.rpt);
  fprintf(fout,"%s",snterm.val);
  if (snterm.tcap)
   { if (buf && strcmp(snterm.buff,"")!=0)
        fprintf(fout,",%s",snterm.buff);
       if (buf && strcmp(snterm.rpt,"")!=0)
        fprintf(fout,",%s",snterm.rpt);
   }
}

void SymbC()

 { SymbTCAP();
  switch (sterm.lex)
   { case ':': Accepter(':');
                Accepter('-');
                fprintf(fout,"%s:-%s",snterm.val,snterm.rpt);
                strcpy(snterm.buff,"");
                if (strcmp(snterm.rpt,"")!=0)
                 { fputc(',',fout);
                   if (buf)
                        strcpy(snterm.buff,snterm.rpt);
                 }
                SymbSB();
                Accepter('.');
                fprintf(fout,".\n");
                break;
       case '.': Accepter('.');
                fprintf(fout,"%s",snterm.val);
                if (strcmp(snterm.rpt,"")!=0)
                  fprintf(fout,":-%s",snterm.rpt);
                fprintf(fout,".\n");
                break;
       default : Erreur();
   }
 }

void SymbSC()
```

```
{ SymbC();
  while (sterm.lex!=FDF)
    SymbC();
}

unsigned int ChargTxt(char *pcl[LTAB])

{ unsigned int i;
  char s[DSTR];

  i=0;
  fgets(s,DSTR,fin);
  while (!feof(fin))
    { pcl[i]=(char *)malloc(strlen(s)+1);
          strcpy(pcl[i],s);
          i++;
          fgets(s,DSTR,fin);
    }
  return i;
}

void NomPred(char *pc,char cl[])

{ unsigned int i;

  i=0;
  while (*(pc+i)!='(')
    { cl[i]=*(pc+i);
          i++;
    }
  cl[i]=FDC;
}

void TriPred(char *pcl[LTAB],unsigned int lt)

{ int i,j;
  char ci[40],cj[40];
  char *ps;

  for (i=1;i<lt;i++)
    { ps=pcl[i];
          NomPred(pcl[i],ci);
          j=i-1;
          NomPred(pcl[j],cj);
          while (strcmp(cj,ci)>0)
            { pcl[j+1]=pcl[j];
              j--;
              if (j>=0)
                NomPred(pcl[j],cj);
              else
                break;
            }
          pcl[j+1]=ps;
    }
}
```

```
struct v *ConstrLstVar(char *cl)

 { char *pc,nv[20];
   unsigned i;
   struct v *lv,*v;

   lv=NULL;
   pc=strchr(cl,'(');
   pc++;
   while (*pc!=')')
     { i=0;
          while (*pc!=',' && *pc!=')')
            { nv[i]=*pc;
              i++;
              pc++;
            }
          nv[i]=FDC;
          if (lv!=NULL)
            { v->sv=(struct v *)malloc(sizeof(Var));
              v=v->sv;
              v->nvar=(char *)malloc(strlen(nv)+1);
              strcpy(v->nvar,nv);
              v->sv=NULL;
            }
          else
            { v=(struct v *)malloc(sizeof(Var));
              v->nvar=(char *)malloc(strlen(nv)+1);
              strcpy(v->nvar,nv);
              v->sv=NULL;
              lv=v;
            }
          if (*pc==',')
            pc++;
     }
   return lv;
 }

unsigned int RechNomVar(char *cl,char *nv,unsigned int j)

 { unsigned int i,lc,ln,eg;
   char *p,c;

   ln=strlen(nv);
   lc=strlen(cl);
   for (i=j,p=cl+j;i<lc-ln+1;i++,p++)
     { eg=0;
          while (eg<ln && *(p+eg)==*(nv+eg))
            eg++;
          if (eg==ln)
            { c=*(p+eg);
              if (i+eg<lc && !isalnum(c))
                break;
            }
     }
   if (i>=lc-ln+1)
     i=lc;
   return i;
```

```
 }

char *UnifCls(struct v *lvu,char *cl)

 { struct v *lvc,*plv;
   char *pc,*pn,*p;
   unsigned i;

   lvc=ConstrLstVar(cl);
   pc=(char *)malloc(strlen(cl)+1);
   strcpy(pc,cl);
   while (lvc!=NULL)
     { i=0;
          while (i<strlen(pc))
            { i=RechNomVar(pc,lvc->nvar,i);
              if (i<strlen(pc))
                { pn=(char *)malloc(strlen(pc)-strlen(lvc->nvar)+strlen(lvu->nvar)+1);
                     strncpy(pn,pc,i);
                     pn[i]=FDC;
                     strcat(pn,lvu->nvar);
                     i+=strlen(lvc->nvar);
                     p=&pc[i];
                     strcat(pn,p);
                     free(pc);
                     pc=pn;
                }
            }
          lvu=lvu->sv;
          plv=lvc;
          lvc=lvc->sv;
          free(plv);
     }
   return pc;
 }

void LstNewVar(struct v *lv)

 { struct v *p;

   p=lv;
   while (p!=NULL)
     { free(p->nvar);
          p->nvar=NewVar();
          p=p->sv;
     }
 }

void UnifNomVars(char *pcl[LTAB],unsigned int lt)

 { unsigned int i;
   struct v *lv,*p;
   char npu[40],npc[40],*pc;

   i=0;
   while (i<lt)
     { NomPred(pcl[i],npu);
          lv=ConstrLstVar(pcl[i]);
```

```
        LstNewVar(lv);
        while (i<lt)
         { NomPred(pcl[i],npc);
          if (strcmp(npc,npu)==0)
           { pc=UnifCls(lv,pcl[i]);
                free(pcl[i]);
                pcl[i]=pc;
                i++;
           }
          else
           break;
         }
        for (p=lv;lv!=NULL;lv=lv->sv,free(p),p=lv)
         ;
    }
}

void SauverTxt(char *pcl[],unsigned int lt)

{ unsigned int i;

  for (i=0;i<lt;i++)
   fprintf(fout,"%s",pcl[i]);
}

void NormPred(void)

{ char *pcl[LTAB];
  unsigned int lt;

  lt=ChargTxt(pcl);
  /*puts("Chargement ok");*/
  TriPred(pcl,lt);
  /*puts("Tri ok");*/
  UnifNomVars(pcl,lt);
  /*puts("Uniformisation ok");*/
  SauverTxt(pcl,lt);
  /*puts ("Sauvegarde ok");*/
}

void main(int narg,char *args[])

{ char nfout[128],*c;

  if (narg<2)
   { puts("Fichier prolog necessaire");
        exit(1);
   }
  else
   { strcpy(nfout,args[1]);
        c=strchr(nfout,'.');
        *c=FDC;
        strcat(nfout,".tmp");
        fin=fopen(args[1],"r");
        if (fin==NULL)
         { printf("Fichier %s indisponible\n",args[1]);
           exit(1);
```

```
        }
      fout=fopen(nfout,"w");
      if (narg==3 && strcmp(args[2],"/n")==0)
       buf=FALSE;
      else
       buf=TRUE;
      nlg=1;
      strcpy(lc,"");
      AnalLex();
      SymbSC();
      fclose(fin);
      fclose(fout);
      fin=fopen(nfout,"r");
      c=strchr(nfout,'.');
      *c=FDC;
      strcat(nfout,".nor");
      fout=fopen(nfout,"w");
      NormPred();
      fclose(fin);
      fclose(fout);
  }
}
```